Getting more from MSX–

with SPECTRA VIDEO and all MSX Computers

SIGMA PRESS

Brian Boyde-Shaw

GETTING MORE FROM MSX

with Spectravideo and All MSX Computers

Brian Boyde-Shaw

Σ
SIGMA
PRESS

ISBN 0 905104 89 7

Published by:

SIGMA PRESS,
5 Alton Road,
Wilmslow,
Cheshire,
UK.

Distributors:

UK, Europe, Africa:
JOHN WILEY & SONS LIMITED
Baffins Lane, Chichester,
West Sussex, England.

Australia:
JOHN WILEY & SONS INC.,
GPO Box 859, Brisbane,
Queensland 40001, Australia.

Printed and bound in Great Britain by
J. W. Arrowsmith Ltd., Bristol

Acknowledgments

CONTENTS

AUTHOR'S NOTE

To write a description of a new computer is a daunting and exciting task, especially one that uses a new and updated dialect of BASIC.

Investigating a new machine can be both surprising and disappointing. Surprising, when you find the new and wonderful things it can do, and disappointing when at the same time you realise that your grandiose ideas do not quite fit in with the capabilities of the language.

Discovering the capabilities of MSX BASIC and the Spectravideo has been exciting, and will no doubt continue to be so for some time yet, but any author of technical books must be honest and admit that rarely is all the work his own.

In my case, this is quite true. I have a number of people to thank for help received during, and before, the writing of 'Getting More from MSX'.

For example, Mick Ellick from Byte Home Computer Club of Nailsea, Avon, who was completely responsible for the initial draft of the final chapter, 'A Change of Face', and for the fundamental research into chapter one, the chapter on editing and debugging programs.

I have also to thank Nailsea Office Equipment, The High Street, Nailsea, from whom I bought the computer, and who repeatedly allowed me access to various other pieces of software to aid my investigations.

Finally, I wholeheartedly thank my long suffering family, Valerie, Charlotte and Hannah, for their tremendous help during the writing, and for allowing me complete peace and quiet for the whole of July and August 1984, Daddy was in computerland and was not to be disturbed! - except to be ushered in for validations and opinions on various aspects of the book as and when required.

Although MSX BASIC goes a long way towards being a complete all round home computer programming language, we have yet to discover the perfect one.

When we do, I hope I will be asked to write the book.

Brian Boyde-Shaw, Nailsea, August 1984.

INTRODUCTION

This book is aimed at filling the need of home computer users who have recently purchased a Spectravideo and, having played the games and experimented with the manual supplied with the computer, now want to find out what exactly the computer is capable of doing by using its built-in BASIC language.

It is also aimed at those interested in the MSX language itself, and wish to become familiar withe the new language - the Spectravideo being just the vehicle for the investigation.

Lastly, it is aimed at owners of other home computers who wish to make a comparison of their machine's BASIC and MSX BASIC. This is one of the main reasons for the unusual layout of the book, in that the description of the language is contained in eight appendices at the end of the book.

The book does not pretend to be a complete description of the Spectravideo BASIC, nor of MSX BASIC, which would probably require a book of two or three times the length of this one. But it does, I hope, give a thorough evaluation of the language, which will encourage readers to investigate further on their own.

The book concentrates on the graphics, including sprites, sound, colour and animation side of the language, as it is the author's opinion, from personal research, that this is what the majority of home computer users will be most interested in initially.

In the main, a problem solving approach is used to produce the programs used to demonstrate the various facilities of the computer, as this, I feel, will remove the abstract approach many books on programming produce.

The book starts with a short introduction to editing and debugging of programs (chapter one) - a facility not dealt with, at the time of writing, by the user's manual supplied with the Spectravideo computer.

Thereafter, the book is divided into six further parts. Part one, chapters two and three, deals with the text and block graphics of the language, and part two, chapter four, with the sprite facility.

Part three, chapters five, six and seven, deal with the graphics commands of the language, while part four, chapters eight and nine, with the musical side.

Part five, chapters ten and eleven, describe some of the special effects that can be achieved with the language. Finally part six comprises eight appendices dealing with the MSX BASIC language itself, which describe, in detail where the previous chapters of the book do not, all the commands, statements and functions in common use, including the mathematical functions of the machine, to cater for those who are not yet overly interested in graphics.

Those of a mathematical leaning will no doubt register that although there are eleven chapters, they are divided between eight parts, 0 to

7, if we include this introduction as part 0!

The appendices are also divided into eight separate but complete parts, making this the first 16 bit home computer book!

CHAPTER ONE
Editing and Debugging

This short chapter is an attempt to give the reader an insight into the screen editing facilities of the computer, with the hope that this will help when it comes to debugging, or finding the errors, in programs.

BASIC EDUCATION

The process of writing most computer programs can be divided into three possible areas, similar to the three R's of basic education, Reading, wRiting and aRithmetic.

With computer programs we can call these three areas wRiting, Reading and Running. Of course wRiting consists of the necessary planning and pre-testing of the proposed program, Reading the listing of parts of the program, and eventually the complete one, and Running the experimental and final running of the program to check it against the aims and objectives set out in the writing part.

As with basic education the three R's are inseparable from each other, and develop together as the final program reaches completion, and again as with one's total education, there is always room for improvement even when the program is complete.

In other words, once the initial ideas of the proposed program have been formulated, the lines of the program have to be written out, usually, if a structured format is attempted, in sections at a time, with each separate section individually tested as the development proceeds.

It is during this development that many of the various 'bugs' appear and have to be dealt with by intelligently reading the error statements that the computer reports to you.

In order to avoid as many of these bugs as possible from cropping up the Spectravideo provides the facility to test out a number of the language's commands and statements before putting them into a program, and this facility is used a great deal in this book, especially in the chapters dealing with the sound facilities.

CURSOR CONTROLS

But before we go on to consider this let us take a look at the cursor controls that the computer provides.

Depending on the model of computer in use the cursor, the white square below the word OK, can be moved in command mode by either the built in joystick or the cursor keys to any position on the screen inside the viewing screen area. The viewing screen area is that area of the screen

between the two border areas at the top and bottom of the screen, and can be changed in colour by the command COLOR,C<ENTER>, where C is any number between 0 and 15. Use of the two symbols <> indicates that whatever key or keys is written between them should be pressed, for example <ENTER> means press the ENTER key.

Moving the cursor by this method results in a CLICK sound from the TV loudspeaker, the same sound that pressing a key produces.

This sound can be removed by typing in CLICKOFF<ENTER>, though this is not advised as it is a good indication that something is indeed happening, and allows the programmer to keep his eyes on the keyboard and not the screen.

To illustrate this facility, practice typing in the statement PRINT first with the CLICK facility off, and then with it on again.

The click sound can be replaced by typing in CLICKON<ENTER>.

Characters from the keyboard can be placed in any position on the screen purely by moving the cursor and pressing a key.

It follows, therefore, that any program line that is called up to the screen by the command LIST, can be edited by moving the cursor to the required position in the line and typing in the necessary letter or letters, and the ENTER key pressed when the line has been amended to suit.

INSERT AND DELETE

Pressing a key will overwrite any character that is already in the cursor position, but extra characters can be inserted into a line of program by first pressing the INS/PASTE key at the point where the characters are to be inserted.

The cursor then changes to a third of its height until it comes out of insert mode by pressing either the INS/PASTE key again, or moving the cursor with the cursor control joystick or keys, or by pressing the ENTER key.

Care must be taken when in the INSERT mode that the joystick is not accidently moved when using the SV 318 model.

Characters can also be deleted from a program line by first positioning the cursor over the character to be deleted and then pressing the DEL/CUT key.

The height of the cursor is not affected in this mode, and pressing the INS/PASTE while in delete mode has no effect except to reduce the height of the cursor.

COMMAND MODE

As mentioned previously numerous programming ideas can be tested

in command mode by using those commands and statements that the computer can react to whilst in that mode.

For example, try the following simple demonstrations to give you some ideas, there is no requirement to go into CAPS LOCK mode unless you wish to:

```
PRINT35<ENTER>
PRINT"BOO"<ENTER>
A=67<ENTER>
PRINTA<ENTER>
A$="BOO AGAIN"<ENTER>
PRINTA$<ENTER>
PLAY"CDEFGAB05C"<ENTER>
FORR=1TO1Ø:BEEP:NEXT<ENTER>
SOUND1,5:SOUND8,15<ENTER>
```

Your screen will not look exactly like this because the results of the various statements will appear on a separate line between the statements you have typed in, together with 'OK' and the cursor.

The last command, the SOUND pair, will require the CTRL/STOP keys to be pressed together to stop the note playing when you've heard enough!

Now is your opportunity to practice using the edit facility, via the cursor controls, to change some or all of the statements and commands in my list, you can call it experimental editing!

But note that lines shunted off the top of the screen will be lost for ever, we are in command mode.

All you need to do to get the statements and commands to produce either a visual or aural result again and again is to move the cursor to any position on the line containing the statement of your choice and press the ENTER key.

No doubt after a little while your screen will be in quite a mess; where, for example, you have pressed the ENTER key on a line NOT containing a statement or command and produced an error statement.

Don't worry, just press the CTRL/L keys together, and the screen will automatically clear and you are ready to start all over again, and improve on your last creation!

Try typing in some decision statements, for example:

```
IFP=1THENPLAY"05C04BAGFEDC"ELSEPLAY"L16AC"
<ENTER>
P=1<ENTER>
P=Ø<ENTER>
```

Now move the cursor around telling the computer what you want by pressing the ENTER key on the correct line and listening to the result.

This is what I mean by experimental editing while developing a program - you have more than twenty lines on screen to scratch around on, consider it then as a visual scratch pad!

DEBUGGING

A program whilst RUNning will normally 'crash', that is, stop running and return to command mode, if it comes across a statement, command or function that it doesn't understand, or discovers that you have failed to do something in the program which must be done before the program can continue.

Usually, an indication of the error it has found is reported to you on the screen in text mode, and, if in high or low resolution mode, your display will be lost.

A certain amount of error trapping is available on the computer, that is, methods of dealing with errors as and when they occur, but I shall only consider those that trap user errors, and these will be dealt with in the chapters that follow, where it is more relevant.

Debugging really refers to correcting errors that the programmer has himself created by one of three things:

1. An incorrect use of the language.

2. A typing error.

3. Leaving out important parts of a statement.

INCORRECT LANGUAGE USE

The first reason will only improve with age!

The more you use the language the more you will understand it and be able to use it correctly, as with any foreign language.

Using the experimental edit idea will greatly increase your awareness of how the language works.

You cannot break anything, neither can you insult the computer with a wrong use of the grammar or syntax, as it is all too easy to do in the early stages of learning say French or German.

The question 'will that work' should always be answered by 'let's try it and see', and then typing your idea into the computer either in command or program mode and studying the results. The method adopted in the chapters of this book will help in that way, as frequently full explanations are given of each line in the computer program used to solve a particular problem.

TYPING ERRORS

The second possible error producing reason, typing errors, can again only improve with age, and I advise the writer of any program, no matter how expert or assured, to RUN the developing program at every available point in it. It is much easier to debug a short program of a few lines

than to try to find the error, or even multiple errors, in a complete program.

Testing of non-text mode program ideas in program mode is quite possible; high and low resolution programs return to text mode when the program ends, by using a 'hold everything' or 'suspended animation' line of:

```
5000 GOTO5000
```

The line number is quite artificial, but should always be greater than the last line of the bit you are testing. To restart you can either press CTRL/STOP and **RUN** it again, or press CTRL/STOP and use GOTO line number, the line number being where you stopped, after, or even before it. The latter is preferable as any variables filled in the program while it was running prior to being held will not be lost, as they would be with the use of **RUN**.

The program can also be put into suspended animation by pressing the STOP key, restarting it by pressing the STOP again.

For example, type in this short program and practice, stopping the ellipse from being filled as often as you like, before the program ends and reverts back to command mode and the text screen by pressing the STOP key on and off :

```
10 COLOR1,15,15:SCREEN1
20 CIRCLE(128,96),70
30 PAINT(128,96)
40 END
```

Now use your newly found abilities to edit the screen by:

1. Changing the SCREEN mode to 2.

2. Altering the numbers in both the **CIRCLE**, **PAINT** and **COLOR** commands.

Don't forget to press the ENTER key after each 'edit' to get the computer to accept your amended program line.

Now see if you can spot the FOUR typing errors in the following program, they can happen quite frequently:

```
10 COLOUR1,15,6:SCREN1
20 CIRCLE(128,96),70
30 PRINT(128,96)
40 END
```

Your first error reported will be:

Syntax error in 10

If you find a mistake in line 10 then correct it, but you will still get:

Syntax error in 10

as there are two errors on this line.

There are syntax errors on lines 20 and 30 as well.

I hope you use the screen editing facilities to correct the mistakes!.

Syntax errors seem to be the most frequent error reported by the computer though there are a number of others, and these are all listed on page 117 to 119 of the computer manual. However number 26 does not work, as a NEXTless FOR is not reported, and is completely ignored by the Spectravideo. A FORless NEXT is reported though.

Error codes 29 and 30 can both be disregarded as both the WHILE and WEND statements are not available on the Spectravideo.

LEAVING OUT IMPORTANT PARTS OF STATEMENTS

The last reason, leaving out an important part of the BASIC language, can sometimes NOT produce an error, for example:

```
PRINT"The second inverted commas are not
required<ENTER>
```

As previously noted a **FOR** without a **NEXT**, and vice versa, will always be reported as an error.

Semi-colons are not required between multiple **PRINT** statements either, for example:

```
A$="bang"<ENTER>
PRINTA$;A$;A$;A$
```

will produce the same result as:

```
A$="bang"<ENTER>
PRINTA$A$A$A$<ENTER>
```

This chapter has been, therefore a short discussion dealing with the various ways of getting your programs to **RUN** correctly, and the chapters that follow will take this discussion further.

Now on to the main part of the book.

CHAPTER TWO
Screen Text

REM; PRINTTAB; LOCATE; LET (assumed use); SCREEN; COLOR; INPUT; FOR...TO...STEP; NEXT; IF...THEN...ELSE; IF...GOTO...ELSE; READ; DATA; GOSUB; RETURN; GOTO; BEEP; MOTOR ON; MOTOR OFF; SOUND ON; SOUND OFF; CLS; ASCII codes; CHR$; Variables; Operators; Punctuation.

In this chapter we shall begin to make the Spectravideo produce text output on the screen.

Text in a computer program usually does one of a number of things.

It can give the necessary information about how to interact with the computer during the program, as far as we are concerned this text is called 'instructions'. Or, it can give the state of play of that interaction, commonly called results, or even more commonly 'the score'. Text is also useful for titles, though more often than not now it is also associated with some sort of graphics, and possibly sound as well.

The first thing we shall look at, therefore, is producing a title screen. So let's start straight away with a short problem.

* *

Write a program to display centrally on the screen a title for a program of your own choice. The whole screen must be the same colour, and preferably have no distractions. Allow the screen to continue to the next part of the program by whatever means you feel are appropriate.

* *

As we are about to write text, then it is reasonable to assume that we should use a text screen. The text screen is obtained by using SCREEN0. This is OK, but the moment the computer RUNs out of something to do, you get the OK prompt and the cursor back on the screen.

This means that the title screen would have at least a cursor in view, as the title would not be the end of any program you were writing, I hope. This is what the problem means by distractions.

The problem asks that the screen should be one colour, that is no discernible border. This means that we shall have to use the COLOR statement, which has three parameters, COLORtext colour,background colour,border colour. In text mode the border is always the same colour as the background, so that part of the problem is easily solved.

Our problem therefore has four parts:

1. Choose the correct screen mode.

2. Choose the correct colour statement.

3. Position the title text.

4. Continue the program.

The only two points left to discuss are positioning the text, and continuing the program. We could, of course, use cursor controls to position the text, similar to the procedure required on less sophisticated computers. But on the Spectravideo, to position anything anywhere on the screen in any screen mode, we use **LOCATE** X,Y, where X is the screen column, and Y is the screen row.

If we want to centralise a piece of text it is usual to count the number of characters in it before using the **LOCATE** statement, and then to calculate the X parameter, or position, by subtracting half the number from half the screen width. This means we could use the formula 'screen width divided by 2, minus (total characters divided by two)', or in computing terms:

```
40/2-LEN(T$)/2
```

which will near enough centralise the text, T$.

This could be done automatically within a program by getting the computer to either **LET** various strings, **T1$**, **T2$**, **T3$** etc., equal the pieces of texts, or to **READ** them from **DATA** statements first.

For example we could say:

```
20 SCREEN0:COLOR4,11,11
30 T1$="GETTING MORE":T2$="FROM
YOUR":T3$="SPECTRAVIDEO"
40 LOCATE20-LEN(T1$)/2,10:PRINTT1$
50 LOCATE20-LEN(T2$)/2,12:PRINTT2$
60 LOCATE20-LEN(T2$)/2,14:PRINTT3$
```

This would print a reasonable title of my choice in the centre of the screen but I would have the cursor there as well, of course. There is no real need to use the command **SCREEN0** here, but I do it to complete the process.

Alternatively we could change line 30 to:

```
30 READT1$,T2$,T3$
```

and add line, say 100, for the DATA

```
100 DATA"GETTING MORE","FROM
YOUR","SPECTRAVIDEO"
```

which would have the same effect.

These two short routines would carry out the necessary positioning calculations automatically for you, but if maths are not your weak point, then the program could be done just as simply by the following routine. I have used large line numbers here as I want to build up a small complete program, and therefore will want to structure my program as near as I can. This means, as we do not have the facility of PROCEDURES in MSX BASIC, I will have to use subroutines instead, calling them by line number instead of by name.

```
1000 REM title screen
1010 SCREEN0:COLOR4,11,11
1020 LOCATE15,10:PRINT"GETTING MORE"
1030 LOCATE16,12:PRINT"FROM YOUR"
1040 LOCATE15,14:PRINT"SPECTRAVIDEO"
1050 RETURN
```

If you type this in and **RUN** it you will get an error, as the computer has at the moment nowhere to **RETURN** to. You can put on line 1045 **GOTO1045**, which will hold the computer program in suspended animation until you press the CTRL and the STOP keys together, and no error will be produced. You could instead start to write the main or control program by typing in **DELETE30-60<ENTER>:**

```
10 REM main program
20 GOSUB1000
99 END
```

and delete lines 30, 40, 50 and 60.

This will remove the error situation, and the computer will have somewhere to go back to now, the **END** statement on line 99.

But we have the problem of the cursor still, which is an intrusion on our otherwise nicely laid out title screen.

We also have another problem, at the bottom of the screen are the five function key windows! More distractions.

So, how do we remove those?

Try changing line 1010 to:

```
1010 SCREEN,0:COLOR4,11,11
```

RUN the amended program, and as if by magic the function key windows disappear, but the screen is not automatically cleared anymore.

When using either **SCREEN0**, **SCREEN1** or **SCREEN2** the screen will automatically clear when the computer reads either one of them in a program, so there is no need to use **CLS**, the clear screen statement. We must add **CLS** to our program as it now stands:

```
1010 SCREEN,0:CLS:COLOR4,11,11
```

or we can write:

```
1010 SCREEN0,0:COLOR4,11,11
```

which will obviously once again clear the screen, but without CLS.

However this still leaves the cursor!

On a graphics screen we do not get a cursor; the cursor is used to indicate where the next piece of text is to go, and as we are assumed to be drawing 'pictures', and not text, on a graphics screen, it is not required.

Line 1010 can now be changed again, to:

```
1010 SCREEN1:COLOR4,11,11
```

CLS is not required, I hope you remember why!

Now RUN this latest addition to our amended routine, and you will see that my three pieces of text are printed very quickly in the top left hand corner of the screen. Why?

A SCREEN1 graphics screen, has a different resolution to a SCREEN0 text screen, and the LOCATE parameters must allow for this.

Each character in a piece of text occupies an area of 6x8 pixels, or dots, like this:

```
* * * * * *
* * * * * *
* * * * * *
* * * * * *
* * * * * *
* * * * * *
* * * * * *
* * * * * *
```

and the LOCATE statement when using SCREEN1 locates a pixel at a time, so LOCATE15,10 means print 15 pixels in from the left hand side of the viewing screen, and 10 pixels down from the top. I say viewing screen now, as this is somewhat different from the TV screen. The viewing screen for text and SCREEN0, and the viewing screen for anything on SCREEN1, are not the same. SCREEN1, or the high resolution graphics screen, as it is called, is slightly wider.

To translate from a text screen to a high resolution graphics screen we must use some more simple mathematics.

SCREEN1 column=SCREEN0((columnx6)+12)
SCREEN1 row =SCREEN0(rowx8)

So we can now amend our subroutine once more, change lines 1020 to 1040 to:

```
1020 LOCATE102,80:PRINT"GETTING MORE"
1030 LOCATE108,96:PRINT"FROM YOUR"
1040 LOCATE102,112:PRINT"SPECTRAVIDEO"
```

If you RUN this amended routine you will indeed see the title printed screen centre, but not for long, as once a program has finished it automatically reverts to the text screen, and as we are PRINTing, on a graphics screen now, the title disappears.

To avoid this, revert to the previously mentioned trick of including a GOTO statement, but in the main program this time:

```
98 GOTO98
```

This will have the effect of keeping the title on the screen until you press the 'CTRL' and 'STOP' keys together.

You can also use the centralising routine:

```
width/2-LEN(T$/2)*6  that is:
LOCATE,256/2-(LEN(T$)/2)*6  or:
LOCATE1,8-(LEN(T$)/2)*6
```

As an interesting side issue, the make up of the screen characters can be shown by using the third screen mode, SCREEN2, which is called the low resolution screen, because everything is displayed in giant size.

Add these few lines to your program:

```
1 SCREEN2:COLOR15,1,1
2 PRINT"D"
3 PRINTTAB(4)"<right graphics>Y"
4 GOTO4
```

When the program is RUN, you will see the letter 'D' in large script, together with the alternate dot graphic rectangle, that is what I mean by <right graphic>, anything in between the < > signs should not be typed in, but the necessary indicated key pressed instead.

You will notice that the 'D' is somewhat hidden, that is why you must add 12 pixels to the previous text to graphics calculation. You will also be able to count the enlarged pixels in the rectangle. SCREEN2 multiplies everything by 6x8, that is, any character is six times wider and eight times higher than in SCREEN0 and SCREEN1.

Remove the four lines 1, 2, 3 and 4 when you have finished experimenting.

Now to carry on with our title screen, that is solving the 'continue the program' part.

To continue the program we must make a choice between doing it automatically or allowing the user to take control.

If the choice is for automation then a suitable time delay will have to be placed in the program at the correct point to enable enough time for the title screen to be read and understood.

It is better, I think, for the user to decide when the next part of the program should start, and therefore the computer will have to be set up to recognise some sort of signal from the keyboard. But this signal

will have to prompted in order that the user will know what to do.

The next line in our short routine therefore could be:

```
1050 LOCATE60,136:PRINT"Do you need
instuctions?"
```

Here the user will have to make a decision and then make that decision known to the computer by pressing a key or typing in an answer:

```
1060 LOCATE96,152:PRINT"Type yes or no":BEEP
```

The BEEP will hopefully get the attention of the user, and the PRINT statement indicates what has to be done, 'yes' or 'no' must be typed in, and it is assumed that the user knows that the ENTER key must be pressed as well. If you feel that this extra information needs to be placed on the screen as well, then it should be included in the PRINT statement on line 1060.

Line 1070 completes the information loop:

```
1070 INPUTANSWER$:BEEP
```

If this routine is now RUN, you will notice that the screen changes back to a text one, complete with the function key windows, and we have lost our title screen, quite upsetting for the user of the program. The reason for this is that the INPUT statement can only be used on a text screen, as INPUTs are usually in text, either letters of the alphabet, numbers or symbols.

We can use the screen statement SCREEN0,0 to remove the function key windows, but the computer is still awaiting an INPUT at the top of the screen.

```
1070 SCREEN0,0:INPUTANSWER4:BEEP
```

To improve matters we can use the INKEY$ function in place of the INPUT statement in line 1070. INKEY$ only recognises one character for each statement, the first one taken from what is called the keyboard buffer, the holding part of the computer's memory that stores key depressions.

So there is no requirement to type 'yes' or 'no', just to press the Y or N key, but this also means that line 1060 must also be changed

```
1060 LOCATE102,152:PRINT"Press Y or N":BEEP
1070 ANSWER$=INKEY$:BEEP
```

We now have either a Y or an N in the computer's memory called ANSWER$, and we must tell the computer what to do with it in order to continue the program.

```
1080 IFANSWER$="Y"ORANSWER$="y"THENGOSUB2000:
     RETURN
1090 IFANSWER$="N"ORANSWER$="n"THENRETURN
     ELSE1070
```

We must make allowances for the fact that the CAPS LOCK key can be operated or not in our routine, and I have used the OR operator to tell the computer to look for both possibilities. The Y or y answer sends the computer to subroutine 2000 to write, we hope, the instructions on the screen, then at the correct time to return to the main program. The N or n answer just sends the computer back to the main program, the instructions are not required.

Using **INKEY$** now in line 1070, does not make the computer change over to a text screen, and so we do not lose our title screen. The use of ELSE 1070 in line 100, ensures that the computer will only recognise Y, y, N or n as a possible value for **ANSWER$**.

We now have a complete routine that displays a title page on the screen and asks the user whether the instructions are needed or not, before continuing with the program.

One thing to be aware of is that, if you press the STOP key, the program will lock, but you may not be aware of it because nothing is happening when you press the Y or N keys. If this occurs try pressing the STOP key and then the Y or N key to see if the program continues. Pressing the STOP key a second time frees the program from the hold or lock position.

By now you may have also realised that the screen and the border do strange things after the short 4 line program, on lines 1 to 4, has been experimented with. This is because calling a screen and then colouring it does not necessarily have the same effect as colouring the display and then calling a screen. To avoid anything untoward happening it is always a good idea to colour first and call the screen second. This means changing line 1010 to:

```
1010 COLOR4,11,11:SCREEN1
```

You may be wondering how, if we remove the function key windows with **SCREEN0,0**, we can get them back again. It is relatively simple to do this, use **SCREEN0,1** or **SCREEN0,2**, or in fact any **SCREEN0,X**, where **X** is any number, to bring them back again. **SCREEN0** will only clear the screen in this instance, its use will not replace the windows.

A second experiment will perhaps indicate the use and abilities of the various **SCREEN** statements:

```
1 SCREEN0:PRINT"screen0":GOSUB7
2 CLS:SCREEN,0:PRINT"screen,0":GOSUB7
3 SCREEN1:PRINT"screen1":GOSUB7
4 SCREEN2:PRINT"screen2":GOSUB7
5 CLS:SCREEN0,1:PRINT"screen0,1 or ,2 etc."
  :GOSUB7
6 END
7 FORD=1TO2000:NEXT:RETURN
8 END
```

Type this short program in and **RUN** it. It will serve to indicate what

Type this short program in and **RUN** it. It will serve to indicate what you get with the various **SCREEN** statements. It will also show what happens to the viewing screen and the border when the **COLOR** statement is not used together with the **SCREEN** statement.

We now come to writing the 'instructions' for our example program that are supposed to be at line 2000, so let's put them there.

We will use the text screen to display the instructions, which means that the **LOCATE** parameters will have to locate characters not pixels.

One facility that the SPECTRAVIDEO has unlike many other computers is that you can play an audio tape at the same time as a program is running, using the dedicated cassette recorder.

To switch the cassette motor on you must use the **MOTOR ON** statement, and **MOTOR OFF** to switch it off. Likewise to switch on and off the audio channel you must use the **SOUND ON** and **SOUND OFF** statements.

So while our instructions are on the screen, we can either play some music to pass the time, or someone's voice previously recorded on tape (and why not your own?), and then play it actually speaking the words.

This is difficult to show in a book, so it's up to you to experiment.

My routine allows for this to happen, and just in case you have chosen to play music, I have programmed the space bar, **CHR$(32)**, to end the playing with **INKEY$**. You can, of course, time the amount of time required to speak all the words of the instructions, and then put in a **FOR....NEXT** delay loop to switch off the cassette **MOTOR** and the cassette SOUND channel at the right time. Here is my routine:

```
2000 REM instructions
2010 COLOR10,4,4:SCREEN0,0:D$=CHR$(31)
2020 MOTORON:SOUNDON
2030 LOCATE14,1:PRINT"INSTRUCTIONS"
2040 PRINTD$;D$;D$;TAB(6)"This book on the
     SPECTRAVIDEO"
2050 PRINTD$;D$;TAB(7)"presents you with
     problems"
2060 PRINTD$;D$;TAB(16)"to solve,":PRINTD$;
     D$;TAB(4)"The book helps you to solve
     them"
2070 PRINTD$;D$;TAB(4)"then extends the study
     by giving"
2080 PRINTD$;D$;TAB(13)"you some more."
2090 PRINTD$;D$;TAB(6)"Press space bar to
     continue."
2100 PRINTD$;D$;TAB(9)
2110 A$=INKEY$
2120 IFA$=CHR$(32)GOTO2130ELSE2100
2130 SOUNDOFF:MOTOROFF:RETURN
```

Why, you may ask, have I used this D$ variable?

To move the cursor down the screen we can use an empty **PRINT**

statement, and used like this it will just print a blank line. But the computer has control characters that actually move the cursor around the screen, and each of these control characters have string variables that we can use in a program called CHR$(X). Every character on the keyboard has a CHR$(X) code we can use, for example the capital letter A has CHR$(65).

The CHR$(X) for 'move the cursor down the screen one row or print line' is CHR$(31). So in line 2010 I have called this character D$, and then used it whenever I wanted to move the cursor down the screen.

You must remember that as it is a character it needs to be PRINTed to be used correctly, so every time it is used in my program it comes after a PRINT statement.

Instead of using LOCATE all the time now, I have LOCATEd the first line of the display and then moved the cursor down with D$, or CHR$(31), and across the screen with the TAB(X) function.

Line 2100 does a special trick with it. We cannot remove the cursor because we are using a text screen, but I have positioned it in the centre of the bottom of the screen, under the last printed line of text, and used it as decoration. Now it is not a distraction, but an ornament!

I have, on line 2120, also used CHR$(32), the space bar character code, to tell the computer only to recognise the space bar as the key to continue the program, by using the IF....GOTO....ELSE statement.

As a final experiment in this chapter, you can now delete the previous experiment from lines 1 to 8 and type in the following lines:

```
1 COLOR15,1,1:SCREEN2
2 FORR=33TO126:PRINT"CHR$";R;CHR$(R)
3 FORD=1TO500:NEXT:SCREEN2:NEXT
4 FORR=150TO215:PRINT"CHR$";R;CHR$(R)
5 FORD=1TO500:NEXT:SCREEN2:NEXT
6 END
```

CHR$(127) to CHR$(149) are blank so we need two FOR....NEXT loops to print out all the characters in the SPECTRAVIDEO's BASIC in large print slowly at the top of the screen.

To look more closely at any particular character, just press the STOP key, then to resume the parade, press it again.

As a fitting end to this second chapter, can you write a program that will display, one at a time, all the available characters in low resolution screen size, but displayed vertically up the screen? The clue is that the screen can hold vertically six characters in SCREEN2 size, before it needs to go back to the top of the viewing screen again.

And can you also amend my last program so that only one FOR....NEXT loop is required.

A clue?

IF you can THEN you R 150 times better than before!

CHAPTER THREE
Gymnastic characters

Graphics on the Spectravideo can be divided into four main areas, block or character graphics, low resolution, high resolution and sprites. As we have seen in chapter two it is possible to mix them, when we placed text on a high resolution screen, SCREEN 1.

Text, after all, is nothing more complicated than specialised block graphics, specialised in that the shapes used represent the alphabet, the numbers 0 to 9, and the necessary symbols required to punctuate both the prose used and the programming statements.

As we also saw in chapter two the graphics characters, or alphanumeric characters, the phrase used to describe both the alphabet and the numbers, is designed on a 6 x 8 grid, and a close study of the last experiment should have shown this. The shape therefore that the character has is entirely up to the designer of the character set supplied in the READ ONLY MEMORY of the computer. This is why the text on some computers looks different on the screen to that on other computers, and similarly the text in books is different too.

In this chapter I want to have a look at the ways we can make use of the computer's built-in graphics characters, those that are accessible from the keyboard by using the left and right graphics keys, together with the letter keys.

I will keep the shape used as a demonstration as simple as possible, and allow you to experiment with whatever fantastic arrangement of characters you may wish. Each graphic shape has a CHR$ code as well, and a table of these will be found in your computer manual. I shall also be showing you how to use these too.

So, for the time being, let us do some more experiments.

To draw a little man in profile we could program into the computer:

```
100 CLS
105 PRINT"<right graphic>H"
110 PRINT"<right graphic>P<left graphic>N"
115 PRINT"<right graphic>P"
120 PRINT"<left graphic>H"
125 PRINT"<left graphic>B"
```

When **RUN** this short program, will produce a 'little man' at the top left hand corner of the screen, together with the **OK** sign and the cursor.

In future, to save writing <right graphic> and <left graphic> I shall write <rg> and <lg> in the listings, so that line 105 would be:

```
105 PRINT"<rg>H"
```

to produce the little man's head.

This short program could also be written as:

```
100 CLS
105 PRINTCHR$(193)
110 PRINTCHR$(201);CHR$(173)
115 PRINTCHR$(201)
120 PRINTCHR$(167)
125 PRINTCHR$(161)
```

if we wanted to make use of the CHR$ codes for the characters.

Naturally, every time we wanted to use our little man in our program we should have to rewrite the lines 105 to 125, which would be a little wasteful in the use of memory. To avoid this we can use assumed LET and some variables, as follows:

```
100 CLS
105 H1$="<rg>H"
110 A1$="<rg>P<lg>N"
115 B1$="<rg>P"
120 L1$="<lg>H"
125 F1$="<lg>B"
130 PRINTH1$
135 PRINTA1$
140 PRINTB1$
145 PRINTL1$
150 PRINTF1$
```

which will again, when RUN, produce our little man at the top of the screen. I have used the first letter of the part of the body that the string variable describes for the five variables codes.

We could also have lines 105 to 125 as follows:

```
105 H1$=CHR$(193)
110 A1$=CHR$(201)+CHR$(173)
115 B1$=CHR$(201)
120 L1$=CHR$(167)
125 F1$=CHR$(161)
```

You will notice the main difference between the two line 110's. Assigning the variable A$ to normal characters by pressing the character keys is the same as assigning text to a variable, you just enclose it all in inverted commas. But to use the CHR$ codes you must 'add' them together with a plus sign. This facility will be used a lot later.

We now have our little man in the computer's memory, with the parts of his body labelled, so that we could use them as often as we wished to in a program.

For example, we could make his legs longer by adding these lines to the program above:

```
146 PRINTL1$
147 PRINTL1$
```

which would make his legs three times longer than before, we have used L1$ three times. You could also make his feet longer by adding this to line 150:

```
150 PRINTF1$;F1$;F1$;F1$
```

a little like Charlie Chaplin, - feet four times as long. Note the use of the semi-colon between each string variable to **PRINT** the 'feet' next to each other on the same screen row.

The program as it stands will only **PRINT** our man in the top left hand corner of the viewing screen, but we may want to put him anywhere on the screen.

To do this we must make use of the **LOCATE** statement, with a **LOCATE** for each **PRINT**. If we only used one **LOCATE** statement, one for the first **PRINT**, then only the head would be printed where we wanted it to be, the computer would start the next **PRINT** statement, the arm, at the beginning of the next row. Our amended program would then look like this:

```
100 CLS
105 H1$="<rg>H"
110 A1$="<rg>P<lg>N"
115 B1$="<rg>P"
120 L1$="<lg>H"
125 F1$="<lg>B"
130 LOCATEX,Y:PRINTH1$
135 LOCATEX,Y+1:PRINTA1$
140 LOCATEX,Y+2:PRINTB1$
145 LOCATEX,Y+3:PRINTL1$
150 LOCATEX,Y+4:PRINTF1$
```

Where X was the column we wanted to place him, and Y the first row for his head, and his arm being placed on Y"1 row, his body on Y"2, etc.

If you attempt to **RUN** this short program now, it will of course 'crash' because the computer does not know the values for X and Y. You could amend line 100 to:

```
100 CLS:INPUTX,Y:CLS
```

which would allow the user to assign values to X and Y before the main part of the program was **RUN** by the computer. Note the use of the comma between the X and Y in the **INPUT** statement, this allows the computer to accept more than a single variable with one **INPUT** statement.

But programming five separate **PRINT** and **LOCATE** statements to produce our little man, as we do in lines 130 to 150 in our program, is again quite wasteful of memory. Once we have assigned the parts of our little man's body into string variables then the next task would be to group his body together as a whole into one string, as follows:

```
M1$=H1$+A1$+B1$+L1$+F1$
```

But if we tried to print this on the screen using PRINT M1$, the computer would do exactly what we had told it to do, it would PRINT our little man's head, arm, body, legs and feet all on the same screen row, next to each other, not quite how we want him!

To get him into one variable, and looking upright and tall as he should be we will have to resort to some trickery.

Once the computer prints out our man's head, the next thing it does is print out his arm next to the head, but we want the arm on the next row and underneath the head. To do this we must tell the computer to do just that, go to the next row and move underneath the head.

The computer has a few CHR$ that carry out this sort of instruction, we saw one in the last chapter.

To get the computer to move the cursor during a program RUN, we can use CHR$(10) to move down one screen row, and CHR$(29) to move backwards across the row, that is from right to left.

We can therefore create a string that tells the computer to do this, for example:

```
R1$=CHR$(10):FORR=1TO2:R1$=R1$+CHR$(29):NEXT
```

This will create a string that tells the computer to move down one row, and then move back across the row two character spaces. This is how R1$ grows as the computer reads that line:

```
R1$=CHR$(10)
R1$=CHR$(10)+CHR$(29)
R1$=CHR$(10)+CHR$(29)+CHR$(29)
```

Why do we only need a loop variable of 2? Well, there are only a maximum of two characters in the parts of the man's body, so the cursor needs only to move back two character spaces each time. But this now brings in another complication to our program. If we attempted to use this trick the parts of his body that only had one character would upset the smooth running of the display. His arm would be printed one space too many to the left, and so on. To avoid this we must make a basic change to our program in lines 100 to 125, by adding an extra character space to those lines that only have one, so:

```
100 CLS:INPUTX,Y:CLS
105 H1$="<rg>H "
110 A1$="<rg>P<lg>N"
115 B1$="<rg>P "
120 L1$="<lg>H "
125 F1$="<lg>B "
130 LOCATEX,Y:PRINTH1$
135 LOCATEX,Y+1:PRINTA1$
140 LOCATEX,Y+2:PRINTB1$
145 LOCATEX,Y+3:PRINTL1$
150 LOCATEX,Y+4:PRINTF1$
```

is the corrected program, and to correct those lines using CHR$ to create

the parts of the body we must do as follows:

```
105 H1$=CHR$(193)+CHR$(32)
110 A1$=CHR$(201)+CHR$(173)
115 B1$=CHR$(201)+CHR$(32)
120 L1$=CHR$(167)+CHR$(32)
125 F1$=CHR$(161)+CHR$(32)
```

which adds a character space, `CHR$(32)`, to each variable.

We can now rewrite `M1$` as folows:

```
M1$=H1$+R1$+A1$+R1$+B1$+R1$+L1$+R1$+F1$
```

which will print each part of the little man's body in the correct place, wherever we may want to print him.

We can now delete the lines that place the little man in a particular place on the screen, lines 130 to 150, and replace them with one LOCATE statement and the M1 and R1 variables, as follows:

```
130 R1$=CHR$(10):FORR=1TO2:R1$=R1$+
    CHR$(29):NEXT
135 M1$=H1$+R1$+A1$+R1$+B1$+R1$+L1$+R1$+F1$
145 LOCATEX,Y:PRINTM1$
```

Don't forget that as we are using the power up screen, SCREEN 0, to print on, we must use the text character LOCATE codes, not the pixel.

We can now place our man anywhere we want to on the viewing screen, but that is all we can do with him at the moment.

It would be a good idea if we could move him about the screen, and create some very simple animation.

Moving About

To do this, once we have placed him somewhere and want to move him on, we must remove him from where he was before.

The simplest way to achieve this is to print spaces in the old position after we have moved him, for example:

`FORD=1TO200:NEXT:LOCATEX,Y:PRINT`some spaces

But where do we get the right amount of spaces from? From:

`M2$=`somes spaces

But how do we create `M2$`? In exactly the same way as we created `M1$`. By making the same character strings, adding them together, but this time filling them with spaces.

Again we can amend our short program to suit:

```
100 CLS:INPUTX,Y:CLS
105 H1$="<rg>H ":H2$="  "
110 A1$="<rg>P<lg>N":A2$="   "
115 B1$="<rg>P ":B2$="  "
120 L1$="<lg>H ":L2$="  "
125 F1$="<lg>B ":F2$="  "
130 R1$=CHR$(10):FORR=1TO2:R1$=R1$+CHR$(29):
    NEXT
135 M1$=H1$+R1$+A1$+R1$+B1$+R1$+L1$+R1$+F1$
140 M2$=H2$+R1$+A2$+R1$+B2$+R1$+L2$+R1$+F2$
145 LOCATEX,Y:PRINTM1$
150 FORD=1TO200:NEXT:LOCATEX,Y:PRINTM2$
```

We can now use this to move the little man around the screen by making line 150 his next position.

Lines using the CHR$ routine would also need to be amended to suit, for example:

```
105 H1$=CHR$(193)+CHR$(32):SP$=CHR$(32)+
    CHR$(32):H2$=SP$
110 A1$=CHR$(201)+CHR$(173):A2$=SP$
115 B1$=CHR$(201)+CHR$(32):B2$=SP$
120 L1$=CHR$(167)+CHR$(32):L2$=SP$
125 F1$=CHR$(161)+CHR$(32):F2$=SP$
```

We now know how to move a sequence of block graphics characters around the screen, but only in a straight line. But they can be moved from left to right, right to left, top to bottom, bottom to top, and diagonally both left to right and right to left.

This can be achieved by either placing the LOCATE and PRINT statements inside a FOR....NEXT loop, or just by using a series of LOCATE and PRINT statements, which either increment the X axis, the Y axis, or both at the same time. To move from left to right we increment the X axis by a plus value, from right to left with a minus value. To move from top to bottom we increment the Y axis with a plus value, and bottom to top with a minus value. To move diagonally we increment both the X and the Y axes at the same time with either plus or minus values depending in which direction we want to go.

We can, of course, use the text screen with SCREEN 0, which clears the screen, but leaves the function windows intact, or we can use SCREEN0,0, which removes the windows, and clears the screen.

We can use the high resolution screen, SCREEN 1, but if we did we would have to use pixel co-ordinates with the LOCATE statement. We would also have to remove the little man each time we moved him by printing background coloured space characters instead of spaces. A high resolution screen, of course, allows for a smoother animation or movement, that is a pixel at a time, instead of, as in SCREEN 0, where movement is one character space at a time.

We could also play around with the low resolution screen, SCREEN 2, using our 'little man', or whatever you have created. Again, we would have to remove him by printing background coloured space characters,

but he would be moving six pixels at a time, the same as one character space at a time, and he would of course be eight times as tall, as well as six times as wide. An interesting screen mode for animating large characters.

Before we develop this block or character graphics idea any further we will pose a small programming problem.

****
Write a program to demonstrate block graphics animation using various screen modes and a block graphics background.
****

One way to write a program is to set out all the mini-problems that solving the main problem throws up. Another way is to write down headings of all the steps that the program must take to solve the problem, and then to write a main program that covers all this, using either procedures or subroutines. As we are unable in MSX BASIC to write named procedures we must resort to subroutines.

We have to give a block graphics demonstration using a block graphics background. This will require two mini-problems, one to create all the strings we may need and the other the background.

But we must decide what sort of a demonstration we are going to give. The best graphics demonstrations are those that involve colour, sound and animation. Be patient, we aren't going into sound yet, and in block graphics each separate colour change will have to be separately programmed. That leaves us with animation. But as it is always best to go forward slowly, I shall keep the animation simple. I shall use a 'little man', and move him around the screen. Animation infers movement in all directions, so how about some gymnastics, hence the title of the chapter. Our little man can run up some steps, to show individual movement in both the X and Y directions at once; jump onto a trampoline, showing continuous movement; bounce off it, more continuous movement, but in the opposite direction. Then coming down again, say after jumping, or not, over a high jump; the or not can be arranged by a random X or Y location. Finally, we can get the little man back to the start position.

So first of all, we'll write the main program to cover all that:

```
4000 REM graphics demonstration program
4010 GOSUB4100:REM create strings
4020 GOSUB4200:REM draw background
4030 GOSUB4300:REM place little man
4040 GOSUB4400:REM display message
4050 GOSUB4600:REM move man
4060 GOSUB4800:REM check and return man
```

Why you ask have I arranged for a message to be displayed, well as it's a demonstration, the user will want to start it when he is ready, so the message should control this.

Line 4060 mentions a check, if the man is going to jump randomly over a high jump, then we shall want to know if he has cleared it or not.

The background must have some steps, a trampoline, and a high jump. The steps can be made from blocks of the square character, CHR$(201), or <rg>P, as can the trampoline, and the sides of the high jump. The little man we already have, but I have also drawn him in both profiles now, the M3$ and H3$ strings etc, being the second profile.

My interpretation of this is as follows and I hope you can follow the slow build up of the steps in lines 4105 and 4110. S5$ is the foot of the trampoline, while S6$ is the canvas, rather thick I'm afraid.

```
4100 REM create strings
4105 SP$=CHR$(201):S1$=SP$+SP$:S2$=S1$+S1$
4110 S3$=S2$+S1$:S4$=S2$+S2$
4115 R2$=CHR$(10)+CHR$(29):S5$=SP$+R2$+SP$
4120 FORR=1TO10:S6$=S6$+SP$:NEXT
4125 H1$="<rg>H ":H2$=" ":H3$=" <rg>H"
4130 A1$="<rg>P<lg>N":A2$=" ":A3$="<lg>N<rg>P"
4135 B1$="<rg>P ":B2$=" ":B3$=" <rg>P"
4140 L1$="<lg>H ":L2$=" ":L3$=" <lg>H"
4145 F1$="<lg>B ":F2$=" ":F3$=" <lg>B"
4150 R1$=CHR$(10):FORR=1TO2:R1$=R1$+CHR$(29):
     NEXT
4155 M1$=H1$+R1$+A1$+R1$+B1$+R1$+L1$+R1$+F1$
4160 M2$=H2$+R1$+A2$+R1$+B2$+R1$+L2$+R1$+F2$
4165 M3$=H3$+R1$+A2$+R1$+B3$+R1$+L3$+R1$+F3$
4170 RETURN
```

To begin with I shall use the text screen to draw the background on, you can then see all the disadvantages with it. I shall make use of the high resolution screen with the same background in the next chapter when we discuss sprites. But this means we must use SCREEN0,0 to remove the function key windows. I shall do this in the main program as, once drawn, we can leave it alone, it's the little man we are interested in. So change line 4020 to

```
4020 CLS:GOSUB4200:REM draw background
```

Lines 4220 and 4230 draw the steps, 4240 and 4250 the trampoline, 4260 to 4280 the high jump, CHR$(193) is not only the man's head, but the thick rope of the high wire.

```
4200 REM draw background
4210 SCREEN0,0
4220 LOCATE3,17:PRINTS4$:LOCATE5,16:PRINTS3$
4230 LOCATE7,15:PRINTS2$:LOCATE9,14:PRINTS1$
4240 LOCATE13,16:PRINTS5$
4250 LOCATE13,16:PRINTS6$;S5$
4260 LOCATE26,17:PRINTS4$
4270 FORR=16TO10STEP-1:
     LOCATE28,R:PRINTSP$:NEXT
4280 LOCATE28,R:PRINTCHR$(193)
4290 RETURN
```

The next job is to place the man on the first step, but as we are drawing him downwards, placing his head first, then the first location must be the position for his head. The first step is X=3, Y=17, he is five characters

tall, therefore he must be five characters higher than the first step, that is Y=17–5=12.

```
4300 REM place little man
4310 LOCATE3,12:PRINTM1$
4320 RETURN
```

Now for the message to start everything going.

```
4400 REM display message
4410 ME$="Press space bar to start":L=LEN(ME$)
4420 LOCATE20-L/2,0:PRINTME$
4430 A$=INKEY$:IFA$<>""THEN4430
4440 A$=INKEY$:IFA$<>CHR$(32)THEN4430ELSE4450
4450 MB$=SPACE$(24):LOCATE20-L/2,0:PRINTMB$
4460 LOCATE3,12:PRINTM2$
4470 RETURN
```

If we are to place messages on the screen in a balanced and attractive way then they should be placed centrally more times than not. To do this I have put the message in a string, ME$, measured the length of it with **LEN(ME$)**, then X **LOCATE**d it by dividing the width of the text screen by two, and then subtracting half the message length from that, see line 4410 and 4420. This method will always place a message in the centre of a screen row within half a character.

Line 4430 ensures that any key that was pressed inadvertantly before the message was displayed, will be ignored, the computer keeps looking in the keyboard buffer until it is empty. Then it moves to the next line and waits for the space bar to be pressed, before first blanking out the message on line 4450, and then blanking out our little man waiting on the first step, prior to moving on up the steps in the next subroutine. As a quick test, can you see anything unnecessary in line 4440? – no prize for getting it right, the answer is at the end of the chapter.

Line 4450 uses the special **SPACE$** function to blank out the message drawn in line 4420.

Now comes the task of moving the litle man across the screen in a gymnastic fashion, that is to run up the steps, jump onto the trampoline, and then bounce over the high jump or not. This is all to be catered for at subroutine 4600, REM move man.

The last thing we did was to remove the man from the first step when the user was ready to start, so now we must place him on the second step, and so on up the stairs. To do this we can use a FOR....NEXT loop, and each time through the loop take a step of two for the X location because each step is two characters wide. To move up the screen at the same time we must increment the Y location by one each time through the loop. As the little man, M1$, is moved up to the next step he must be removed from the previous one by blanking him out with M2$, with a short delay between the two operations. We can do all this in a couple of lines after writing the delay subroutine:

```
4900 REM delay routines
4910 FORD=1TO400:NEXT
4920 FORD=1TO200:NEXT
4930 FORD=1TO100:NEXT
4940 RETURN
```

Using this arrangement of delay loops we can get various time delays with only one routine, all depending where we enter it. 4900 - 700 units, 4920 - 300 units and 4930 - 100 units.

Now to move the little man:

```
4600 REM move man
4610 Y=11:FORX=5TO9STEP2:LOCATEX,Y:PRINTM1$:
     GOSUB4900
```

which declares where the man must be first of all, Y=11, on the second step, and starts the loop going.

```
4620 LOCATEX,Y:PRINTM2$:Y=Y+1:NEXT
```

which removes the man, and increments the Y location for the next step.

Now the man must run off the top step and eventually jump down onto the trampoline. The run can be achieved with another `FOR....NEXT` loop:

```
4630 FORX=10TO17:LOCATEX,9:PRINTM1$
4640 GOSUB4930:LOCATEX,9:PRINTM2$:NEXT
```

X is now at 17 and Y is at 9. To get the man to drop down onto the trampoline we can `LOCATE` and `PRINT` him just above it, if we put him actually on it we would have removed part of the trampoline when we moved him on, and the whole screen would have to be redrawn, we will save this trick for later.

```
4650 LOCATE18,10:PRINTM1$:GOSUB4930:
     LOCATE18,10:PRINTM2$
```

Now for the random part of the whole program, the distance he actually jumps and whether he clears the high jump or not. The maximum number of changes of the Y location to get the man to clear the jump is ten, the height of the jump plus stand, so we need a random number generated between 1 and 10 each time to bounce the man off the trampoline.

```
4660 N=INT(RND(-TIME)*10)+1
```

This number for Y's eventual position can then be put in a `FOR....NEXT` loop which increments the LOCATE statement up and across the screen, both for Y and X. The next position for X is 19, and for Y it's 11, so:

```
4670 X=19:FORY=11TONSTEP-1:LOCATEX,Y:
     PRINTM1$:GOSUB4930
4680 LOCATEX,Y:PRINTM2$:X=X+1:NEXT
```

will bounce our man to a random height.

What goes up must come down, so he must now fall to the ground with:

```
4690 FORY=NT017:LOCATEX,Y:PRINTM1$:
     LOCATEX,Y-1:PRINTM2$:NEXT
```

This loop prints the new man, and immediately goes back and blanks out the old one without any delay. This way he rapidly falls to the ground. But he will remove part of the background as he drops, as we are reprinting the character spaces, so we must now redraw the screen as soon as he lands, but as he landed and was immediately blanked out we must also reprint him, with:

```
4700 LOCATEX,18:PRINTM1$:GOSUB4200
4710 RETURN
```

We must now communicate with the user once more, telling him the results of the jump, and asking him if he wants to do it again. For this we will need three subroutines, one for good jumps, one for the bad, and one to get the man back to the start of run.

```
4500 REM fail message
4505 ME$="Hard luck, try again? Y or
     N":L=LEN(ME$)
4510 LOCATE20-L/2,0:PRINTME$
4515 A$=INKEY$:IFA$<>""THEN4515
4520 A$=INKEY$:IFA$<>"Y"ANDA$<>"y"ANDA$<>
     "N"ANDA$<>"n"THEN4520
4525 RETURN
```

Using this routine will write the fail message over the previous message at the top of the screen, but it is just that bit shorter. To overcome this messy arrangement, we can blank out the first message as soon as the user has started the program by adding line 4405 to the 'display message' routine:

```
4405 LOCATE0,0:PRINTSPC(40)
```

which blanks out the whole row, forty characters.

Notice that in line 4520 the computer is trapped to only accept either upper or lower case Y or N, but also that when ‹›, 'is not equal to', is used we must use the operator AND and not OR to give the computer a choice of decisions to make.

We can now also write a similar win message routine:

```
4550 REM win message
4555 ME$="Well done, try again? Y or
     N":L=LEN(ME$)
4560 LOCATE20-L/2,0:PRINTME$
4565 A$=INKEY$:IFA$<>""THEN 4565
4570 A$=INKEY$:IFA$<>"Y"ANDA$<>"y"ANDA$<>
     "N"ANDA$<>"n"THEN 4570
4575 RETURN
```

We now have to check the X position of our little man to see if his

bounce was in fact a win or a fail, and if the user wants to do it all again. The high jump is located at X=28, see line 4270, so any X position after the jump greater than 28 will be over the rope and therefore a win, anything less a fail:

```
4800 REM check and return man
4810 IFX<29THENGOSUB4500:IFA$="y"ORA$="Y"
     GOTO4830ELSE4870
4820 IFX>28THENGOSUB4550:IFA$="y"ORA$="Y"
     GOTO4830ELSE4870
```

Here we have used 'IF A$=' and can therefore use the OR operator for multiple choices.

If the decision is to do the run again then the little man must be got back to the steps, if not then we can end the program:

```
4830 LOCATEX,18:PRINTM2$:REM blank out man
4840 FORR=X-1TO1STEP-1:LOCATER,18:PRINTM3$:
     GOSUB4930
4850 LOCATER,18:PRINTM2$
```

These two lines will turn the man round to face the other way, using M3$, and then move him back across the bottom of the screen, starting at the random X position minus 1. We have to use R for repeat, in the FOR....NEXT loop here for the X location, as X is already being used for the random X location.

Once he is at the left hand side of the screen we must turn him round again and put him on the bottom step to wait for the space bar to be pressed again.

Our main program stopped at this point with the 'check and return man' subroutine at line 4060, so we must now take some action as a result of the user's decision:

```
4070 LOCATE1,18:PRINTM2$:IFA$="Y"ORA$="y"
     GOTO4030
4080 GOSUB4900:CLS:SCREEN,1
4090 END
```

Line 4070 will start the whole routine again, whereas line 4080 will, after a short delay, clear the screen and put the function key windows back on the screen. I have designed some of the programs in this book to be part of a big demonstration, that is why this routine started at line 4000, so line 4090 could just as well be:

```
4090 RETURN
```

making the routine at 4000 a subroutine.

This short program has shown the use of SCREEN 1 together with block graphics, and the following program shows the use of animation with block graphics on SCREEN 2, the low resolution screen:

```
100 SCREEN2
105 LOCATE120,64:PRINT"<rg>H"
```

```
110 LOCATE96,96:PRINT"<rg>0<rg>P<rg>L"
115 LOCATE120,128:PRINT"<rg>P"
120 LOCATE96,160:PRINT"<rg>0<spc><rg>L"
125 FORD=1TO300:NEXT:CLS
130 LOCATE120,32:PRINT"<rg>H"
135 LOCATE96,64:PRINT"<rg>C<rg>P<rg>E"
140 LOCATE120,96:PRINT"<rg>P"
145 LOCATE96,128:PRINT"<rg>C<spc><rg>E"
150 LOCATE0,160:PRINT"<rg>P<rg>P<rg>P<rg>
    P<rg>P<rg>P<rg>P<rg>P<rg>P<rg>P"
155 FORD=1TO300:NEXT:CLS
160 A$=INKEY$:IFA$=""GOTO105
165 IFA$<>""THENCLS:END
```

On the low resolution screen you can of course locate anywhere on the screen exactly as you can with **SCREEN 1**, the high resolution screen, but to position characters or blocks so that they do not overlap each other, it is best to assume that the screen is divided up into nine character positions on the X axis, and six character positions on the Y axis, though the last X position may put your character somewhat outside your screen.

The locations for this are as follows:

X=0, 24, 48, 72, 96, 120, 144, 168, 192
Y=0, 32, 64, 96, 128, 160

You will notice that on the X axis the increment is 24, that is 6 times 4, and on the Y axis 32, 8 times 4.

The reason for this is that in the low resolution mode each pixel in the character is blown up or magnified by a factor of four, so as a character is six pixels by eight pixels normally, it will now occupy a twenty four by thirty two pixel area instead.

If you didn't want to place your first character, a space or character, at location 0 in either axis, then to avoid overlapping, you would have to increment each position by either 24 for the X axis, and 32 for the Y axis.

In my short demonstration program, I have started at the 0 location in both X and Y axes, and have printed a big man doing some exercises, jumping up and down moving his arms and legs. The program alternates between two pictures, with a short delay between each picture, where the man has to jump up to avoid the carpet being rolled out underneath him each time.

Lines 160 and 165 allow the program to be ended by pressing any key.

In the next chapter we will see how much better it is to use sprites, although a little more complicated, to get objects to move around the screen. Block graphics will still be used though to draw the background.

CHAPTER FOUR
Sprite Characters

Block characters are all right in their place, and can be quite large, or as large as you want to make them, but they are to a degree quite chunky, and moving them requires a clumsy blanking out procedure.

Sprites on the other hand can be shaped to suit your requirements. They occupy a slightly larger area on the screen than a single block character, in fact either an 8 x 8 pixel block, or a 16 x 16 pixel block, remember block characters occupy only a six by eight pixel area. And moving them is simplicity itself, it's all built in to the sprite facility, as we shall soon see.

In order to use the sprite facilities we must first design our sprite, remembering that we can have up to 32 different ones at any one time, and all on the screen at the same time too, each one on a different plane, as it is called. Naturally you also can also use the same sprite pattern over and over again in any one of the 32 planes.

So the first thing to do is design a sprite. I am going to use much the same screen 'little man' character that I used in the last chapter, you can design them how you wish, the variations are such that there are 768 possible full-size patterns available. The manual describes one method for producing sprites and I shall look at this first, in order to explain how sprite patterns are designed and created. But there is another method, easier, in my opinion, that we shall discuss later.

Designing a Sprite

As the sprite is made up of eight rows of eight bits of information, we must naturally have 8 x 8, or 64, bits of information somewhere that the computer can use. These are stored in **DATA** statements in the program, 64 characters, or bits of data, for each different sprite, and it is usual to put these in eight **DATA** statements at the end of the program.

Each row of **DATA** has to be **READ** into the sprite character string called **SPRITE$(S)**, where S is any number from 0 up to 255, the number you have called the sprite. If we are going to **READ DATA** eight times, then a **FOR...NEXT** loop is necessary to do this eight times, but the method is somewhat strange in that the sprite string is made up by stringing together the binary value of each **DATA** statement by using the expression:

```
S1$=S1$+CHR$(VAL("&B"+DA$))
```

where:
DA$ is the information in each **DATA** statement,
&B tells the computer that what follows is a binary number,
VAL tells the computer to find the VALue of what follows,
CHR$( ) tells the computer to find the CHaRacter STRING of what

follows, and,
S1$ is the name given to that particular sprite pattern.

Our little example program on how to create a sprite looks like this:

```
10 COLOR1,7,7:SCREEN1
20 FORT=1TO8:READDA$
30 S1$=S1$+CHR$(VAL("&B"+DA$)):NEXT
40 SPRITE$(1)=S1$
50 PUTSPRITE0,(128,96),1,1
60 GOTO60
100 REM data for little man
110 DATA00011000
120 DATA00011000
130 DATA00010000
140 DATA00011100
150 DATA00011000
160 DATA01111000
170 DATA01001000
180 DATA00001100
```

Line 40 makes the first sprite, called SPRITE$(1), from the information available in memory called S1$. We could, of course, have made the program shorter by calling S1$, SPRITE$(1), straightaway, in line 30 by:

```
30 SPRITE$(1)=SPRITE$(1)+CHR$(VAL("&B"+DA$)):
   NEXT
```

and leaving out line 40, but then it is easier to call some other sprite, say SPRITE$(2) for example, the same pattern, by making it equal to S1$ as required, then by saying now SPRITE$(1)=SPRITE$(2).

So where did I get the DATA from for the sprite pattern?

As I mentioned earlier a sprite is made up from an 8x8 grid or matrix, exactly as in figure 1, the squares you do not want are called 0, and those you do are called 1. So as our little man, now with his leg up, running, is as in figure 2, you can see how the DATA statements in lines 110 to 180 are arrived at.

To create a sprite, first draw your blank matrix, or photocopy mine in Figure two, then fill in the squares that you need to create your pattern, then fill in the 0's and 1's, next put the binary numbers in the DATA statements, and, lastly, put the information into the computer's memory with lines similar to 20, 30 and 40 in my short program.

Line 50 just places the sprite in the centre of the screen, similar to a LOCATE statement, and tells the computer which plane you want the sprite on, 0 in this case, colours the sprite black, the first 1, then tells the computer which sprite to put at this location, in this case a 1 again.

You can in, fact, put more than one sprite on one plane, although it's not usual, because the computer will put one sprite on that plane, remove it, then put the second sprite there, then remove that, and so on dealing with all the sprites you have programmed to be on that plane. This means

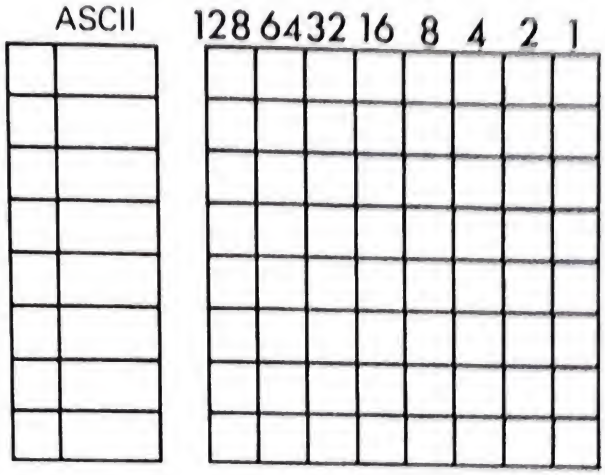


Figure 1

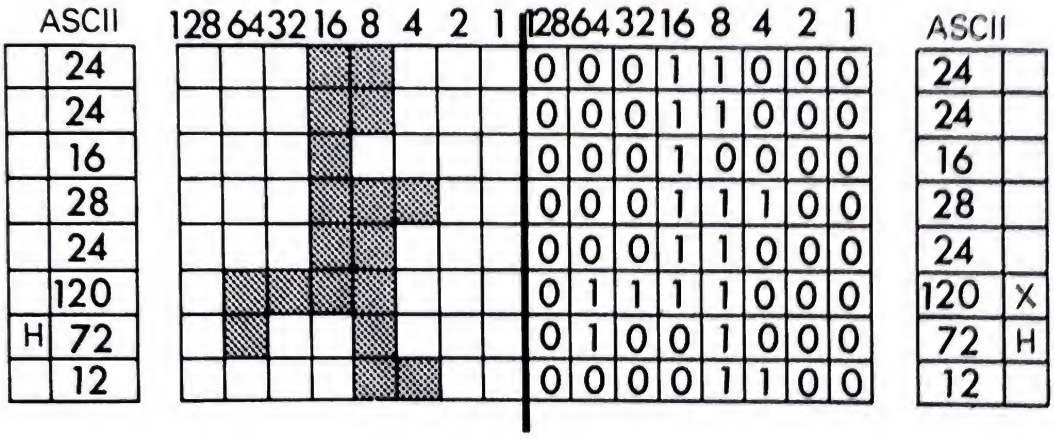


Figure 2

that all the sprites will be flashing on and off, at a rate determined by the number of sprites you have on any one plane. I do not recommend this method though, unless you want this type of special effect.

The same sprite can be placed on different planes, and in different colours, purely by changing the plane number, the parameter P, immediately after `PUTSPRITE`, and the colour number, C, immediately after the location parameters in `PUTSPRITEP,(X,Y),C,spritenumber`.

Running this short program will produce a very small black 'little man' sprite character in the centre of the screen.

CHARACTER STRING SPRITES

Now for the 'quick and easy' way of producing sprites in MSX BASIC on the Spectravideo.

Looking at line 30, repeated here, from my program, you will notice that the sprite variable S1$ is in fact a string made up of characters, or `CHR$s`, which has a value depending on the binary number in the eight `DATA` statements all added together.

```
30 S1$=S1$+CHR$(VAL("&B"+DA$))
```

This means that in actual fact S1$ is:

```
S1$=CHR$(VAL("&B"+00011000))
+CHR$(VAL("&B"+00011000))
+CHR$(VAL("&B"+00010000))+ etc.
```

It is usual to write `CHR$s` with a decimal number, for example `CHR$(65)` means "A", and `CHR$(66)` means "B", that is why we had to write "`&B`" to tell the computer that it was reading a binary number. And by the way the binary number was also a string, notice the inverted commas in the brackets around `&B`, and the `READ DA$` in line 20, not `READ DA` as it would be for a true numerical variable. This is why we had to use the function `VAL` to find its value before it could be put into the `CHR$` function, which only uses number variables, not strings.

From the above it is reasonable to suppose that S1$ could be written as a series of `CHR$s` added together, but written in the usual fashion, that is with decimal numbers inside the brackets.

So how do we convert the binary number into a decimal in order to do this?

The eight numbers in each `DATA` statement in lines 110 to 180, are in fact a ' byte' of information, and each number is called a 'bit' of that byte. Each bit in a byte has a decimal value depending on its position in the binary number, and each bit can be either a 0 or a 1. This is what is meant by 'binary', a counting system with only two numbers, 0 and 1, 'bi' means two. In decimal we have ten numbers 0 to 9, as 'dec' means ten.

The equivalents are as follows:

binary number	1	1	1	1	1	1	1	1 *one byte - 8 bits*
decimal number	128	64	32	16	8	4	2	1

This means that if our binary number has 1's in each bit then the decimal equivalent number is:

128+64+32+16+8+4+2+1 or 255

If any of the bits are 0, then that decimal bit number is not counted.

For example line 110 has:

```
110 DATA00011000
```

This means that we have a decimal value of:

binary number	0	0	0	1	1	0	0	0
decimal number	0	0	0	16	8	0	0	0 = 16+8 = 24

Similarly we can convert the other seven DATA statements into decimal numbers, as follows:

line120 00011000 = 24
line130 00010000 = 16
line140 00011100 = 28
line150 00011000 = 24
line160 11110000 = 120
line170 01001000 = 72
line180 00001100 = 12

We can now rewrite S1$ as follows:

```
S1$=CHR$(24)+CHR$(24)+CHR$(16)+CHR$(28)
    +CHR$(24)+CHR$(120) +CHR$(72)+CHR$(12)
```

This means that our short sprite program is now even shorter:

```
10 COLOR1,7,7:SCREEN1
20 S1$=CHR$(24)+CHR$(24)+CHR$(16)+CHR$(28)
   +CHR$(24)+CHR$(120)+CHR$(72)+CHR$(12)
30 SPRITE$(1)=S1$
40 PUTSPRITE0,(128,96),1,1
50 GOTO50
```

This can now be written using **READ** and **DATA** statements as before by changing line 20 and adding line 100:

```
20 FORR=1TO8:READDA:S1$=S1$+CHR$(DA):NEXT
100 DATA 24,24,16,28,24,120,72,12
```

and our sprite creation program is simplicity itself.

This again will place our little man in the centre of the screen.

Using this method it is still advisable to draw out the sprite pattern in the 8x8 matrix, and then add up the decimal equivalents to find the CHR$ numbers required or the numbers for the DATA statements.

One further advantage of this method is that we can replace many CHR$ codes with their ASCII equivalents straight from the keyboard. For example, CHR$(65) we know is "A", and CHR$(66) is "B", a full list of the CHR$ codes is given on pages 113 to 115 of the user's manual, but note that CHR$(32) to CHR$(35) are incorrect. CHR$(32) is a space or blank, and the other three move down one.

This means it could be possible to create a sprite of your own design by just using the keys, though in my experience it is somewhat unlikely. Anyway, look at this even shorter program for creating a sprite:

```
10 COLOR1,7,7:SCREEN1
20 S1$="AAAAAAAA":SPRITE$(1)=S1$
30 PUTSPRITE0,(128,96),1,1
40 GOTO40
```

This will create a sprite made up from two parallel vertical lines, because as A=ASCII code 65, or CHR$(65), we have 65=64+1, that is in binary 01000001, repeated eight times like this:

```
01000001
01000001
01000001
01000001
01000001
01000001
01000001
01000001
```

This short program could be made even shorter by combining lines 20 and 30, as follows:

```
10 COLOR1,7,7:SCREEN1
20 SPRITE$(1)="AAAAAAAA"
30 PUTSPRITE0,(128,96),1,1
40 GOTO40
```

Or could even be written on just two lines as follows:

```
10 COLOR1,7,7:SCREEN1:SPRITE$(1)="AAAAAAAA":
   PUTSPRITE0,(128,96),1,1
20 GOTO20
```

What could be simpler than playing around with a lot of DATA statements, if all you really want to do is practice using the sprite facillty?

This now means that our previously shortened program, at lines 10 to 50, can have its CHR$s replaced with the key equivalents where this is possible:

```
10 COLOR1,7,7:SCREEN1
20 S1$=CHR$(24)+CHR$(24)+CHR$(16)+CHR$(28)
   +CHR$(24)+"xH"+CHR$(12)
30 SPRITE$(1)=S1$
40 PUTSPRITE0,(128,96),1,1
50 GOTO50
```

DON'T FORGET THAT THE KEYS MUST ALWAYS BE INSIDE INVERTED COMMAS, it's all part of a string. Lower case 'x' is `CHR$(120)`, and upper case 'H' is `CHR$(72)`. Any of the keys can be used, including the punctuation, numbers and graphics, but using `READ` and `DATA` statements would create some difficulty.

Our little man is very small, as basic sprites are only made up from an 8x8 pixel matrix block, but there are a number of ways to make them bigger. An exciting one is to combine four matrix blocks of 8x8 together, and create what is known as a magnified sprite, but with a more detailed design. A planning chart for this is in the Screen and Sprite Appendix, but we could say that instead of:

```
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
```

we can have this, four blocks of 8x8 bits:

```
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
1111111111111111
```

Each block of 8 x 8 bits must be calculated separately and then programmed into the sprite strings as before, but in a particular order, after drawing your 16x16 pattern out using the block matrix pattern, see Figure 5.

The four lots of information are programmed in alphabetical order as follows:

```
A   C
B   D
```

That is, the left hand two first and the right hand two last.

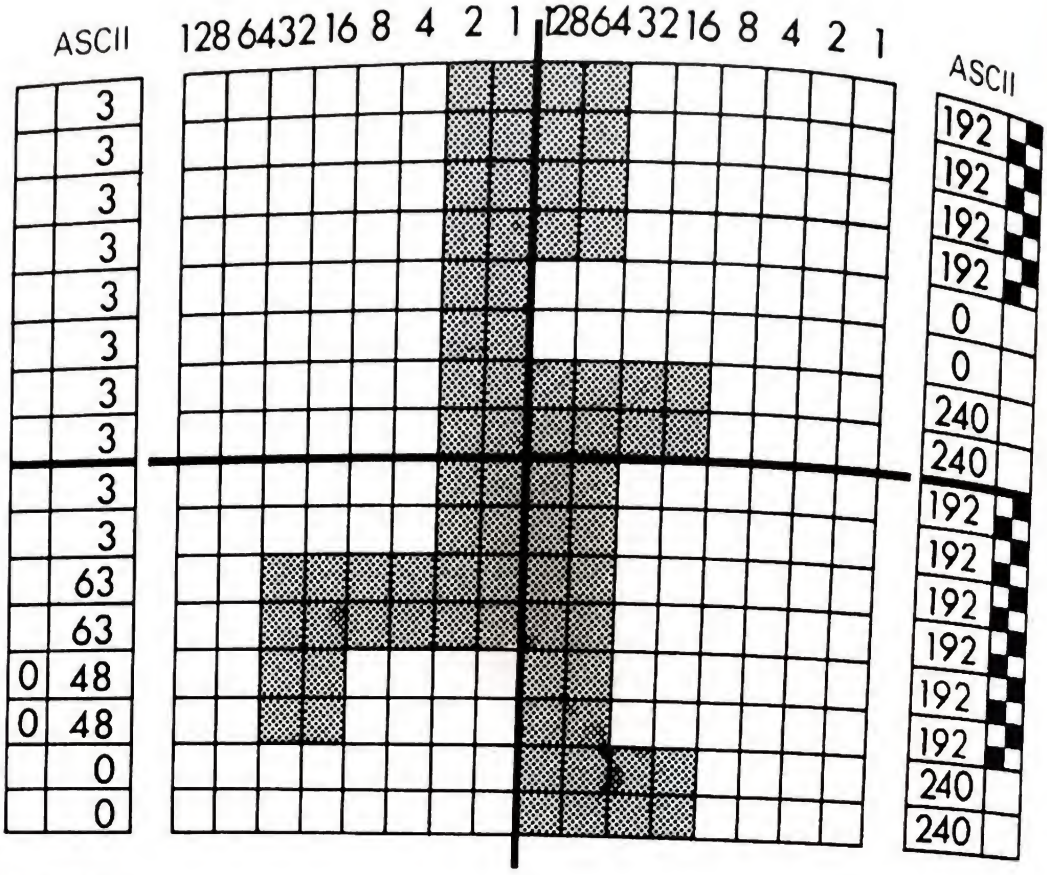


Figure 3

We must also change the SCREEN statement somewhat, by adding a 2 to it:

```
SCREEN1,2
```

And we can only use numbers up to 63 for the sprite numbers when we are using magnified sprites.

This tells the computer to expect four lots of `DATA`, and give a magnification of two.

You can, of course, `READ` into the computer 32 lines of `DATA` statements, copying the blocks of eight data bits straight into the statements, but I think the shorter method is better, which means you can either create four strings, say `SA$`, `SB$`, `SC$` and `SD$`, and then add them all together, `S1$=SA$+SB$+SC$+SD$`. Or you can create only two strings, one for the left hand pair of blocks, `AB$`, and one for the right hand pair, `CD$`, remembering to get them in the correct order, and then add these two together to form `S1$`, `S1$=AB$+CD$`.

Figure four shows you my creation for my magnified little man, and the CHR$ codes required to program him. Remember I have deliberately kept my sprite pattern as simple as possible to aid understanding, you can make them as involved as you wish.

You will notice in the program that follows that I have used `CHR$(Ø)` to get a blank string of eight zeros, you won't find this `CHR$` in the table, but the computer will accept it and realise what you mean. More experienced users of BASIC are warned not to use `CHR$(256)`.

```
10 COLOR1,7,7:SCREEN1,2
20 AB$=CHR$(3)+CHR$(3)+CHR$(3)+CHR$(3)
   +CHR$(3)+CHR$(3)+CHR$(3)+CHR$(3)
   +CHR$(3)+CHR$(3)+CHR$(63)+CHR$(63)
   +CHR$(48)+CHR$(48)+CHR$(0)+CHR$(0)
30 CD$=CHR$(192)+CHR$(192)+CHR$(192)
   +CHR$(192)+CHR$(0)+CHR$(0)+CHR$(240)
   +CHR$(240)+CHR$(192)+CHR$(192)
   +CHR$(192)+CHR$(192)+CHR$(240)+CHR$(240)
40 S1$=AB$+CD$:SPRITE$(2)=S1$:REM or
   SPRITE$(2)=AB$+CD$
50 PUTSPRITE0,(128,96),1,1
60 GOTO60
```

The only limitation to putting all the sprite information into one string from the start is that one line of BASIC is restricted to only 255 characters. The maximum number of CHR$(XXX) you could get into one line is twenty eight, but if the CHR$s did not all contain three numbers, and there are a lot that don't, then it would indeed be possible sometimes. It's up to you to experiment, but remember that the longer a line of BASIC is, the more difficult it is to understand.

We can of course put all the sprite information into the computer using READ and DATA statements, that is, replacing lines 20, 30 and 40 with the following, it is a much simpler and shorter method when dealing with the bigger sprites:

```
20 FORR=1TO32:READDA:S1$=S1$+CHR$(DA):NEXT
30 DATA 3,3,3,3,3,3,3,3,3,3,63,63,48,48,0,0,
   192,192,192,192,0,0,240,240,192,192,192,
   192,240,240
40 SPRITE$(2)=S1$
```

This short program when RUN will now place a magnified little man sprite in the centre of the screen.

You can make the sprites even bigger by joining magnified ones together, albeit on different planes, this is how explosions can be simulated, starting with, say, four sprites joined together, moving them around together and then moving them apart after the explosion.

To demonstrate this let us alter the previous program to suit.

```
10 COLOR1,7,7:SCREEN1,2
20 FORR=1TO32:READDA:S1$=S1$+CHR$(DA):NEXT
30 DATA 3,3,3,3,3,3,3,3,3,3,63,63,48,48,0,0,
   192,192,192,192,0,0,240,240,192,192,192,
   192,240,240
40 SPRITE$(2)=S1$
45 SPRITE$(3)=S1$:SPRITE$(4)=S1$:
   SPRITE$(5)=S1$
50 PUTSPRITE0,(128,96),1,2
55 PUTSPRITE1,(144,96),2,3:PUTSPRITE2,
   (128,112),3,4:PUTSPRITE3,(144,112),5,5
60 FORD=1TO3000:NEXT
70 PUTSPRITE0,(20,0),1,2:FORD=1TO200:NEXT
```

```
80 PUTSPRITE1,(200,0),2,3:FORD=1TO200:NEXT
90 PUTSPRITE2,(20,174),3,4:FORD=1TO200:NEXT
100 PUTSPRITE3,(200,174),5,5
110 GOTO110
```

This program when **RUN** will place four little men on the screen, but not quite in the middle, as the top left hand pixel of the top left hand sprite is programmed to be located in the middle of the screen, 128,96 is the middle.

After a short delay, in line 60, the four men will jump apart and position themselves in the four corners of the screen, each one after a short delay. If I had used 192 as the Y location code for the two sprites at the bottom of the screen, then they would have disappeared from view, as a Y code of 192 would have positioned the top left hand pixel at the bottom of the screen. Similarly the code 0 in the X position would mean that half the sprite would be missing on a normal television screen, so be careful. These positioning difficulties must be allowed for when deciding what location codes to use to position your sprites.

You can now see that we could have a single sprite made up from the whole 32 sprites available on screen at any one time, and each one magnified. The programming of their movement could be a little clumsy, but it is quite possible. You will probably also have noticed that there was no requirement to remove the sprite from the last position it occupied, as I said before this is automatic, once the PUTSPRITE statement has been changed.

This brings us nicely onto the programming requirements to make our sprites move around the screen. Movement can be achieved by two methods, first by jumping from one location to another as in the last program, or secondly by putting the **PUTSPRITE** statement into a **FOR....NEXT** loop, and incrementing either the X or the Y location, or both.

For example, amending the previous program again will show this second method:

```
10 COLOR1,7,7:SCREEN1,2
20 FORR=1TO32:READDA:S1$=S1$+CHR$(DA):NEXT
30 DATA 3,3,3,3,3,3,3,3,3,3,63,63,48,48,0,0,
   192,192,192,192,0,0,240,240,192,192,192,
   192,240,240
40 SPRITE$(2)=S1$
45 SPRITE$(3)=S1$:SPRITE$(4)=S1$:
   SPRITE$(5)=S1$
50 PUTSPRITE0,(128,96),1,2
55 PUTSPRITE1,(144,96),1,3:PUTSPRITE2,
   (128,112),1,4:PUTSPRITE3,(144,112),1,5
60 FORD=1TO3000:NEXT
70 PUTSPRITE0,(20,0),1,2:PUTSPRITE1,(200,0)
   ,1,3:PUTSPRITE2,(20,174),1,4:PUTSPRITE3,
   (200,174),1,5
80 FORD=1TO3000:NEXT
90 FORX=21TO128:PUTSPRITE0,(X,0),1,2:NEXT
100 FORY=1TO96:PUTSPRITE0,(128,Y),1,2:NEXT
```

```
110 FORY=1TO96:PUTSPRITE1,(200,Y),1,3:NEXT
120 FORX=199TO144STEP-1:PUTSPRITE1,(X,96),
    1,3:NEXT
130 Y=173:FORX=21TO128:PUTSPRITE2,(X,Y),1,4:
    Y=Y-1:IFY<112THENY=112
140 NEXT
150 FORY=173TO0STEP-1:PUTSPRITE3,(200,Y),1,5:
    NEXT
160 FORX=199TO20STEP-1:PUTSPRITE3,(X,0),1,5:
    NEXT
170 FORY=1TO174:PUTSPRITE3,(20,Y),1,5:NEXT
180 Y=173:X=21:FORR=1TO100:PUTSPRITE3,(X,Y),
    1,5:Y=Y-1:X=X+1:NEXT
190 Y=72:X=122:FORR=1TO39:PUTSPRITE3,(X,Y),
    1,5:FORD=1TO25:NEXT:Y=Y+1:X=X+1:NEXT
200 FORX=162TO144STEP-1:PUTSPRITE3,(X,112),
    1,5:FORD=1TO50:NEXT:NEXT
210 GOTO210
```

Each line of this program moves a sprite in a particular direction, and I have made them black in each PUTSPRITE statement help understand what is happening in the program. You will also note that each sprite has the same pattern, but you could, of course, make them all different, and I have written the program as if they were. If you use sprites all of the same pattern, then you can use the same sprite number and just place each one you want to use on a different plane, for example in line 70, which places the four sprites in the corners of the screen, I have removed the time delays, and the line can be written:

```
70 PUTSPRITE0,(20,0),1,2:PUTSPRITE1,
   (200,0),1,2:PUTSPRITE2,(20,174),1,2:
   PUTSPRITE3,(200,174),1,2
```

In this way the same sprite has been used four times, SPRITE$(2), but has been placed on different planes. You could in fact have 32 sprites on the screen, all the same, and then mix and match them as you please until you have 32 different sprite patterns on the screen all at the same time. Sprite graphics are indeed a complex and involved subject, and this chapter will only whet your appetite to experiment a lot more.

Now to explain what is happening.

Line 10 - chooses the colours for the text, (not being used), or foreground, (not needed here, sprites are independent colourwise), for the background and for the border and then selects the high resolution screen.

Line 20 - defines the pattern for the left hand side of the sprite, `AB$`.

Line 30 - defines the pattern for the right hand side of the sprite, `CD$`.

Line 40 - defines the whole sprite pattern as `S1$`, and defines `SPRITE$(2)` as having the pattern of `S1$`.

Line 45 - defines `SPRITE$(3)`, `SPRITE$(4)` and `SPRITE$(5)` as having the same `S1$` sprite pattern.

Line 50 - locates sprite 2 on plane 0, in colour black, colour code 1, at location X=128, Y=96.

Line 55 - locates the other three sprites, 3, 4 and 5, on planes 1, 2 and 3 respectively, also in black, at various X and Y locations in the central area of the screen.

Line 60 - delay loop.

Line 70 - Locates the four sprites in the four corners of the screen.

Line 80 - delay loop.

Line 90 - moves sprite 2 along the X axix, horizontally along the top of the screen, 107 pixels.

Line 100 - moves sprite 2 down the screen, back to its original position in the central area.

Line 110 - moves sprite 3 vertically down the screen.

Line 120 - moves sprite 3 horizontally across the screen back to the central area.

Line 130 - moves sprite 4 in a diagonal direction up the screen towards the central area. When it arrives at location Y=112, it then moves in a horizontal direction across the screen back to its original central area position.

Line 140 - a **NEXT** is needed to be placed here and not on the previous line, as it would not be read until Y=112, and therefore prior to that the program would crash as the **NEXT** would be hidden.

Line 150 - moves sprite 5 vertically up the right hand side of the screen.

Line 160 - moves sprite 5 horizontally across the top of the screen.

Line 170 - moves sprite 5 vertically down the screen to the bottom.

Line 180 - moves sprite 5 diagonally up the screen towards the central area.

Line 190 - moves sprite 5 diagonally down the screen, showing that one sprite can pass completely over another without changing the display, and that sprites can be made to move slower than normal with the addition of a time delay loop.

Line 200 - moves sprite 5 back to its original position in the central screen area at a slower rate still.

Line 210 - keeps the program from ending, until a CRTL/STOP is executed.

We have seen now how sprites move, and how they can be moved horizontally, vertically and in both directions at the same time, by the use of **FOR....NEXT** loops, and changing the X and Y location variables within them by using either negative or positive increments.

It should also be noted that every time a PUTSPRITE statement is used, any or all of the parameters in it can be changed, including the plane location, colour and sprite number.

I mentioned and demonstrated that sprites could pass over each other, see line 190 in the last program, and this depends on their priority, that is, which sprite plane they are in, the lower the number the higher the priority. This means that sprites in plane 0 move over the top of any other sprites in the other planes. If you didn't quite see the sprite priority then change the colour of the sprite in line 190 to white, colour code 15, and RUN the program again.

The selection of the correct plane is, of course, important if you don't want particular sprites to disappear when they cross each other's paths.

The fact that sprites can cross each other's path has another advantage, that is, they can also be shown to collide when they do cross, and hits can be registered in the computer's memory. This occurs when a switched on bit of one sprite coincides pixel location wise with a switched on pixel of another sprite. To use this facility we use the ON SPRITE GOSUB statement, which gives the computer somewhere else to go if a collision occurs, a subroutine in fact. But it must be enabled by using the statement SPRITE ON just before it is required.

It is a little limiting in that any collision between any sprites will be noticed, but this can be overcome to some extent by an intelligent use of the SPRITE ON and SPRITE OFF statements. To terminate the use of the ON SPRITE GOSUB facility in a program completely use the statement SPRITE STOP.

To demonstrate the use of these interrupt control statements, as they are called, due to the fact that they interrupt the flow of the program from its normal direction, in a similar way to IF....THEN statements, add the following lines to our last program:

```
5 ONSPRITEGOSUB500
185 SPRITEON
195 SPRITEOFF
205 SPRITESTOP
500 CLS
510 RETURN
```

Here's the explanation of them:

Line 5 - enables the facility for detecting sprite collisions, and tells the computer which subroutine to go to for post-collision action. It is best to put this at the start of the program, as we do not need to switch the facility on until we need it.

Line 185 - switches on the sprite collision facility.

Line 195 - switches off the sprite collision facilty, not strictly necessary in this program as no other collision will occur in the program, but it is always good practice to switch it off when it is not required.

Line 205 - disables the collision facility altogether.

Line 500 - the subroutine here just clears the screen, no need to be complicated in a demonstration. But it could be used for updating a score, or creating some other screen effect as a result of the collision.

Line 510 - returns the computer to where it detected the collision, and carries on with the program from where it left off.

You will perhaps notice that at the moment you can only have one subroutine for collisions, as there is only room for one in the **ON SPRITE GOSUB** statement. But it is possible to set a flag, or indicator, so that other routines can be used depending on what exactly happened either prior to, or at, the time of the collision.

To do this you can use a **IF....THEN....ELSE** statement in the subroutine, or a **ON....GOSUB**, or **ON....GOTO**, providing care is taken never to **RETURN** to the original program before **RETURN**ing to the first subroutine and **RETURN**ing from there. Without proper care and experienced control you could get your program in quite a mess, so take great care when you feel the need to extend the ONSPRITEGOSUB facility in this manner.

Sprites can, of course, be used in **SCREEN 2** mode in exactly the same way as they are used in **SCREEN 1**.

So it is quite possible to give, for example, the big man in the SCREEN 2 program in the last chapter, blue eyes by creating two circular sprites in a subroutine, and calling them from lines 122 and 147, making the locations different to suit for each part of the program. The extra lines may look like this:

```
122 GOSUB200:PUTSPRITE0,(X,Y),4,0:
    PUTSPRITE1,(X,Y),4,1
147 GOSUB200:PUTSPRITE0,(X,Y),4,0:
    PUTSPRITE1,(X,Y),4,1
200 REM create sprite eye
210 S1$=CHR$(24)+CHR$(60)+CHR$(126)+CHR$(231)
    +CHR$(231)+CHR$(126)+CHR$(60)+CHR$(24)
220 SPRITE$(0)=S1$:SPRITE$(1)=S1$
230 RETURN
```

Can you rewrite line 210 making use of **READ** and **DATA** statements, you will have to use an extra line as well.

It is up to you to calculate your own X and Y locations, your idea of where to place the eyes may be different to mine!

A second method of obtaining magnified sprites is to place a 1 after the screen command, instead of a 2, **SCREEN1**. This means that the computer is asked to magnify the normal small 8x8 pixel sprite to a 16x16 magnified size. But with this method, as you will see, the sprites therefore contain less detail, they have only 64 pixels now instead of 256, as they have with **SCREEN2**. I suppose these could be called 'low resolution magnified sprites'. Here is a resume of the sprite types:

SCREEN1,2 or **SCREEN2,2** — high resolution magnified sprites, 16x16=256 programmable pixels.

`SCREEN1,1` or `SCREEN2,1` — low resolution magnified sprites, 8x8=64 programmable pixels.

`SCREEN1,0` or `SCREEN2,0` — normal size sprites, 8x8=64 programmable pixels.

You can leave the 0 out of the `SCREEN1,0` and `SCREEN2,0` commands and the sprite will automatically be of the normal unmagnified size. 0 is the default sprite value of the `SCREEN1` and `SCREEN2` commands.

The following program demonstrates the versatility of sprites, it is quite short and well worth the time taken to type it in. I will explain it line by line, and how to use it as well:

```
5 REM Sprite demo
10 COLOR1,7,7:SCREEN0,0
20 CLS:LOCATE2,2:PRINT"size: 0, 1 or
   2";INPUTSIZE:IFSIZE<0ORSIZE>2GOTO20
30 CLS:LOCATE2,2:PRINT"colour: 1 to 15";:
   INPUTCOLOUR:IFCOLOUR<1ORCOLOUR>15OR
   COLOUR=7GOTO30
40 CLS:LOCATE2,2:PRINT"height: 1 to 32";:
   INPUTHEIGHT:IFHEIGHT<1ORHEIGHT>32GOTO40
50 COLOR1,7,7:SCREEN1,SIZE
60 FORR=1TOHEIGHT:S$=S$+CHR$(127):NEXT:
   SPRITE$(1)=S$
70 PUTSPRITE0,(128,96),COLOUR,1
80 KEYON:ONKEYGOSUB100,110
90 GOTO90
100 RUN
110 END
```

Line 20 asks the user to type in the size of sprite wanted, either normal, a 0, magnified normal, a 1, or magnified, a 2.

Line 30 asks the user for the colour that the sprite must be displayed in, a number from 0 to 15, but it won't accept colour code 7, as this is the background colour of the screen, and if this colour, cyan, were used the sprite would be invisible.

Line 40 asks the user for the 'height' of the sprite, this will determine how many bytes of sprite information is `READ` from the DATA, which are all filled in spaces, `CHR$(127)`. Experiment with this particularly, and see just how many different sized sprites you can actually get.

Line 50 selects the high resolution screen, but you can change this to the low resolution, it won't make any difference.

Line 60, this line `READ`s all the necessary `DATA` to correspond to the `HEIGHT` input of line 40.

Line 70 places the sprite in the centre of the screen.

Line 80 switches on and enables the Function Key select facility, and provides two illegal subroutines for the computer to go to when either function keys 1 or 2 are pressed. The command `ONKEY` cannot use `GOTO`,

we therefore have to cheat a little to get the program to work in a simple fashion.

Line 90 holds the screen display in suspended annimation until a function key is pressed.

Line 100 **RUN**s the program again, and allows fresh choices to be made.

Line 110 **END**s the program.

When you have typed the program in, **RUN** it, and choose the various parameters you wish to see displayed concerning the sprite. Then when you want another choice press function key 1, or press function key 2 to end the program.

And so now to the problem for this chapter.

****

Amend the problem program in chapter three so that a sprite can be used instead of a block graphics character for the little man.

****

The first thing to do is to reorganise the graphics demonstration program around the use of sprites. The subroutines at lines 4300, 4600 and 4800 need to be changed as they deal with the block graphics character. Also the subroutines that draw the background and display the messages must be amended as we are now using **SCREEN 1**, the high resolution screen, and all the locations will be incorrect.

I shall deal with each subroutine in turn.

REM draw background

Obviously line 4210 must indicate a change of screen and a different coloured background, if we are to use one, so:

```
4210 COLOR1,7,7:SCREEN1,2
```

will suffice.

All the locations must now be changed using the formula we discussed previously, bearing in mind that where a location is used in a **FOR....NEXT** loop, these variables will have to be amended as well.

For example lines 4220 and 4270 could be:

```
4220 LOCATE30,136:PRINTS4$:LOCATE42,128:
     PRINTS3$
4270 FORR=128TO80STEP-1:LOCATE180,R:
     PRINTSP$:NEXT
```

REM place little man

This routine must now also be used to create the little man sprite, so will contain the lines we used previously to create the magnified sprite, but note that we cannot now use **S1$** for the sprite pattern store as

we have used this for the space string in the create string subroutine, therefore use some other variable name, say SA$, or go straight into calling the sprite with SPRITE$(2)=AB$+CD$, as suggested in the REM statement on line 40 in that program.

Instead of the LOCATE and PRINT statements on line 4310 we can now use:

```
4350 PUTSPRITE0,(30,96),15,2
```

which places a little white man on the first step. The change in line number makes room for the four lines required to create the sprite.

REM display message

Again the locations will have to be amended, especially the lines that place the message in the centre of the screen, bearing in mind that the total number of pixels across the screen is 256, therefore the number required for the centralisation routine on line 4420 is a half of 256, or 128. But if we subtract half the length of the message we shall not have the correct location even now, as the function LEN(ME$) only counts the number of characters in the message, not the number of pixels. We must therefore multiply the variable L we get by 6, and then add 12 as before. Line 4420, for example, will now be:

```
4420 LOCATE128-((L*6)+12)/2,0:PRINTME$
```

Line 4450 will also have to be similarly changed.

REM move man

In this routine we can use one PUTSPRITE statement to replace both the LOCATE and PRINT statements on each line, calculating the new locations each time to allow for the use of SCREEN 1, remembering to recalculate the STEPs as well!

Line 4660 generates a random number to decide how high the little man will be able to jump, but is calculated in characters. We must therefore change this also to allow for pixels. Each character is eight pixels high, therefore the random number generated must be multiplied by 8 to use it in the FOR....NEXT loop for the Y direction.

Therefore:

```
4660 N=INT(RND(-TIME)*10+1):N=N*8
4670 X=126:FORY=88TONSTEP-1:PUTSPRITE0,(X,Y),
     15,2:X=X+1:GOSUB4930:NEXT
```

Line 4680 will of course be deleted.

REM check and return man

This routine is similar to the previous one, in that all the PRINT and LOCATE statements can be replaced with PUTSPRITE statements, the new X and Y locations calculated, as should the limits for X in lines 4810 and 4820. It will also be a good idea to define a new sprite, one with the little man facing the opposite way, but I will leave that up to you.

REM fail message, and
REM win message

These two routines will have to have their X and Y locations recalculated in order to print out the message.

One final reminder, don't forget to remove all the lines that blank out the little man block characters, sprites do this automatically every time their location is changed.

If you have persevered to the end of this problem, you should now have a block graphics background, drawn on a high resolution screen, using magnified sprites to do the gymnastics.

In the next chapter we shall be looking at one of the ways to draw backgrounds in high resolution graphics mode.

CHAPTER FIVE
Draw Strings

Built into MSX BASIC and the Spectravideo is an extra graphics or drawing system based on the **DRAW** statement. This allows any shape of any size to be built up on the screen, even circular ones if you have the patience, though other BASIC graphics commands can do this quicker and better, so it is best to use this facility for drawing straight lines. Other graphics commands, such as **PAINT**, **GET** and **PUT**, can be used to operate on the graphics picture produced with the **DRAW** statement, to produce quite interesting effects, and it is this facility that we shall be looking at in this chapter.

Basically, the **DRAW** statement allows specific shapes to be built up on the screen in **SCREEN 1** mode, though some very interesting effects can be obtained in the low resolution mode, but their description is outside the range of this particular book.

The **DRAW** statement is followed by a string expression containing all the information required to draw a particular shape, but this information should not exceed 255 characters as such. There are ways of overcoming this, by building into the expression other strings that have been previously defined in the program.

Using this facility does require some small knowledge of co-ordinate geometry, but don't let this put you off, providing you can draw shapes on a piece of graph paper, all should be well. Planning in advance, as with most forms of programming, is all that is required to produce even the most complicated shapes and graphics screens.

Lines can be drawn in any direction by using various single letter codes together with numerical values to indicate the distance that the line has to move. The codes are as follows, where the lower case 'n' is the numerical distance to be moved, and 'x' and 'y' are the normal location codes:

Un	- move vertically upwards.
Dn	- move vertically downwards.
Ln	- move horizontally to the left.
Rn	- move horizontally to the right.
En	- move diagonally up and to the right.
Fn	- move diagonally down and to the right
Gn	- move diagonally down and to the left.
Hn	- move diagonally up and to the left.
Mx,y	- move to a particular screen location using high resolution

mode location codes. Moves may be relative, that is, with respect to the previous last location, or absolute, starting from the top left hand corner of the screen again. Placing a '+' sign before the 'x' location code will force a relative move, otherwise the move will be absolute.

B - move to the new location, but do not plot any points on the screen, that is, will not draw a line.

N - move, draw a line if required, and then return to the original position before the move was made.

Ad - sets an angle to move through before plotting or drawing the line, 'd' has the following values and parameters:
0 - 0 degree, no angular movement.

1 - 90 degree, a right angle.

2 - 180 degree, two right angles.

3 - 270 degree, three right angles

Cc - sets a particular colour for drawing the lines, until changed by another C code number, 'c' has the following values and colours:
0 - transparent, draws in background colour.

1 - black.

2 - medium green.

3 - light green.

4 - dark blue.

5 - light blue.

6 - dark red.

7 - cyan, a bluish tinge.

8 - medium red.

9 - light red.

10 - dark yellow.

11 - light yellow.

12 - dark green.

13 - magenta, darkish pink.

14 - grey.

15 - white.

Sr - sets a scale factor, 'r' may have a value from 0 to 255, but r divided by four, r/4, is 'f', the actual scale factor, which is multiplied by the 'n' value given in the U,D,L,R,E,F,G,H and relative M commands to give the actual distance moved. The default value, the value used if no code 'r' is used, is 1, that is no-scaling, and will be the same as using a command of S4, (f=4/4=1).

X(string variable); - tells the computer to draw a shape according to the information contained in the string variable X$, which must have been previously defined elsewhere in the program.

In any of the foregoing DRAW commands the arguments n, x, y, c, and r can be either a constant, a given number, or a numeric variable, such as A or B for example, and could therefore be used to move a shape around the screen, as will be seen later.

To draw a black empty SQUARE shape in the top left hand corner of the screen, type in this short program:

```
10 COLOR,7,7:SCREEN1
20 DRAW"C1BM15,10D100R64U100L64"
100 GOTO100
```

Line 10 selects a cyan background screen and a cyan border, but notice that no text or foreground colour has been selected, there is no number before the first comma. It is best to select colours within the **DRAW** command itself.

Line 20 selects the colour black, then executes a blank move, a move without any drawing taking place, to position x=15, y=10, then draws the square. Try changing the R and L codes to 100, and you will get a rectangle drawn.

Now change line 20 to:

```
20 DRAW"C1BM15,10D100C4R64C6U100C15L64"
```

This will give the square four different coloured sides, black, dark blue, dark red and white.

Now add line 30:

```
30 A$="D100BL32BU50R64":DRAW"BR32XA$;"
```

This line defines the variable A$ as a set of movements that draw a cross, but uses the blank move code 'B' to move first left, BL, and then up, BU. Then a DRAW command is executed to first blank move right 32 points before using the variable A$. Notice how the A$ variable has a 'X' before it and a ';' after it. When the program is RUN, the square is drawn and then divided into four by the crossed lines. Notice also how the lines are drawn in white, white being the last colour code used before line 30, a C15 code.

If we now wish to change the colour of the cross in the square, we

must make a change to line 40:

```
30 A$="D100BL32BU50R64":DRAW"BR32C1XA$;"
```

By adding 'C1' to the DRAW command the cross will now be black.

Now change line 20 as follows:

```
20 DRAW"BM15,10S1C1D100C4R64C6U100C15L64"
```

Adding 'S1' to the DRAW command on line 20 has reduced the size of the square by a factor of 16. Each line has been reduced by a factor of four, which means we can get 16 small squares inside the big one.

Now add line 40:

```
40 DRAW"BM15,10S2C1D100C4R64C6U100C15L64"
```

which is the same as line 20, except that the scaling factor is a half instead of a quarter, S2 has been used instead of S1. You should now have a square four times as large as the line 20 one, or four times as small as the original one, which we shall now put back with line 50:

```
50 DRAW"BM15,10S4C1D100C4R64C6U100C15L64"
```

You should now have three squares in the top left hand corner of the screen, all replicas of each other except for the cross. Line 30 draws the cross, and of course draws it in the smallest square, as it follows line 20 which draws the smallest square.

To put the cross into the other two squares we must repeat line 30 after lines 40 and 50:

```
45 A$="D100BL32BU50R64":DRAW"BR32C1XA$;"
55 A$="D100BL32BU50R64":DRAW"BR32C1XA$;"
```

Naturally as we are drawing crosses three times in this short program we can put line 30 into a subroutine, and replace the original lines 30, 45 and 55 with:

```
30 GOSUB200
45 GOSUB200
55 GOSUB200
200 REM draw cross routine
210 A$="D100BL32BU50R64":DRAW"BR32C1XA$;"
210 RETURN
```

You will also notice that the outside lines of the smaller squares have been overdrawn by the bigger square as it is drawn, so that they are now all black.

If you wish to watch the action taking place then you can add the following:

```
25 GOSUB300
35 GOSUB300
42 GOSUB300
47 GOSUB300
```

```
52 GOSUB300
300 REM time delay
310 FORD=1TO500:NEXT
320 RETURN
```

Here is a somewhat compacted version of this demonstration program:

```
10 COLOR,7,7:SCREEN1
20 DRAW"BM15,10S1C1D100C4R64C6U100C15L64"
30 GOSUB300:GOSUB200:GOSUB300
40 DRAW"BM15,10S2C1D100C4R64C6U100C15L64"
50 GOSUB300:GOSUB200:GOSUB300
60 DRAW"BM15,10S4C1D100C4R64C6U100C15L64"
70 GOSUB300:GOSUB200
100 GOTO100
200 A$="D100BL32BU50R64":DRAW"BR32C1XA$;"
210 RETURN
300 FORD=1TO500:NEXT
310 RETURN
```

You will also notice that the final square, the largest and the original, must now have a scaling factor in its DRAW string otherwise the scaling factor previously used would be acted upon.

This program can now be further amended and reduced to demonstrate one other facility of the **DRAW** command. The line which draws the square is also used three times, but each time with a different scaling factor. We can make the scaling factor a variable, 'R', and use a subroutine to draw the square, as follows, amending the compacted program:

```
20 R=1:GOSUB400
40 R=2:GOSUB400
60 R=4:GOSUB400
400 REM drawing square routine
410 DRAW"BM15,10S=R;C1D100C4R64C6U100C15L64"
420 RETURN
```

Notice how the variable 'R' is used followed by a semi-colon, that is: S=R;, this means 'the scaling factor S equals the numeric variable R'.

The semi-colon indicates to the computer that what follows are normal DRAW commands not associated with the variable 'R'. It is used in the same way with '**XA$;**' in line 200.

For the variable 'R', either a lower case or an upper case letter can be used, the computer will recognise them both.

The full program could now look like this:

```
10 COLOR,7,7:SCREEN1
20 R=1:GOSUB400:GOSUB300:GOSUB200:GOSUB300
30 R=2:GOSUB400:GOSUB300:GOSUB200:GOSUB300
40 R=4:GOSUB400:GOSUB300:GOSUB200:GOSUB300
100 GOTO100
200 A$="D100BL32BU50R64":DRAW"BR32C1XA$;"
210 RETURN
```

```
300 FORD=1TO500:NEXT
310 RETURN
400 DRAW"BM15,10S=R;C1D100C4R64C6U100C15L64"
410 RETURN
```

Lines 20, 30 and 40 assign a value to the variable 'R', then send the computer to the routine that draws the square at line 400. Next a short delay at line 300, and then the cross is drawn at line 200, using the previous scaling factor to fit it into the already drawn square. In this short **DRAW** command demonstration program most of its facilities have been demonstrated, with the exception of N and A. A is an advanced form of U, D, L and R, where lines can be drawn at the four angles from any point on the screen, which may have been drawing a line at some particular angle by the use of the Mx,y command.

N allows lines to be drawn from a central point, by returning the start point back to the beginning of the previous line each time.

The second part of this chapter gives a further demonstration of the use of the **DRAW** command, together with the associated advanced graphics commands, **PAINT**, **GET** and **PUT**.

The following short program draws a small square using the DRAW command, then colours it in using magenta. Then the computer takes the square and places it in another part of the screen.

```
10 COLOR,7,7:SCREEN1
20 DRAW"BM112,71C13D50R32U50L32"
30 PAINT(113,72),13
40 DIMA(10,10)
50 GET(112,72)-(144,122),A
60 PUT(150,125),A,PSET
70 GOTO70
```

Line 10 chooses the colours for the background and border.

Line 20 draws a square at location x=112, y=72 in colour 13, magenta.

Line 30 fills in the square with the same colour, 13, using the PAINT command.

Line 40 creates or dimensions an area in memory called an array which reserves sufficient space to store whatever you are going to tell the computer to put there.

Always make sure you have reserved enough memory space or your program will crash, you can always reduce the amount in the array until you have the minimum. We'll discuss this in greater detail later.

Line 50 takes an area of the graphics screen and places it in the array dimensioned in line 40. It is usual to declare the area of the graphics screen required by mentally drawing a diagonal line across the area required and reading the end location, this is then placed in the second bracket. The first bracket takes the start location.

Line 60 places or puts this array of graphic screen information in a

particular place on the screen determined by the two co-ordinates, X=150, Y=125. The command PSET ensures that the graphics array is displayed in the correct colour, that is, the colour it was copied in.

Other commands and operators are available and will be dealt with in a later chapter. Leaving out the operator or argument altogether allows the array to be displayed in a complementary colour.

Line 70 stops the program from ending until a CTRL/STOP is used.

The following full demonstration program shows how the DRAW command can be used, together with PAINT, GET and PUT to draw a house, garage, bushes and flowers.

```
10 DIMW(10,10):DIMD(10,10):COLOR,4,4:SCREEN1
20 H$="R120D110L120U110"
30 DRAW"BM78,50S4C6XH$;"
40 PAINT(80,52),6
50 DRAW"BM90,60S1C10XH$;"
60 PAINT(95,65),10
70 GET(90,60)-(120,105),W
80 PUT(161,60),W,PSET
90 PUT(90,110),W,PSET
100 PUT(161,110),W,PSET
110 DRAW"BM128,100C13S4R25D60L25U60"
120 PAINT(130,102),13
130 GET(128,100)-(153,160),D
140 LOCATE145,134:PRINT"<rg>H"
150 LOCATE139,109:PRINT"7"
160 DRAW"BM70,50C12E70F70L140"
170 PAINT(75,48),12
180 PUT(20,100),D:PUT(45,100),D
190 B$="U6L10U2L5U3L4U4R6U2R5U3R4U5R12D2
    R2D5R2D3R8D3R2D4L6D3L12D6L4"
200 DRAW"BM225,170C2XB$;"
210 DRAW"BM175,190C2XB$;"
220 PAINT(226,169),2:PAINT(176,189),2
230 F$="C12U10C=P;NU2NE2NR2NF2ND2NG2NL2NH2"
240 P=10:DRAW"BM82,180XF$;"
250 P=13:DRAW"BM+8,15XF$;"
260 P=15:DRAW"BM+10,5XF$;"
270 P=13:DRAW"BM+20,15XF$;"
280 GOTO280
```

Line 10 saves enough space for the two arrays H and D, colours the border and background dark blue, and selects the high resolution screen mode.

Line 20 defines H$ as a rectangle 120 pixels wide x 110 pixels deep, this will be used for the house.

Line 30 draws the house, at location x=78, y=50, with a normal scaling factor of 1. The scaling factor must be used, because if the program is run a second time, as it is sure to be, then the house will be drawn with the last scaling used. The house will be drawn in colour 6, dark red. A colour code must be used in the DRAW command if the house

is to be filled in, otherwise when the colour is specified in the PAINT command, the house will not be filled, but the rest of the screen will be. Also, you cannot specify a different coloured outline for your shape to the filling, they must both be the same.

Line 40 fills in the house in dark red. Notice that the start point for the fill or paint must be inside the area to be filled, otherwise the screen will be filled instead. It is best to drop inside the area by one or two pixels in both the X and Y positions and use that location. The PAINT command can lead to a lot of confusion if care is not taken over this and the previously mentioned fact that the border and the fill colour must always be the same, and always declared.

Line 50 draws a smaller version of the house, a window, scale used S1, or one sixteenth, with a dark yellow border at location x=90, y=60.

Line 60 fills in the window in dark yellow.

Line 70 takes the area indicated by the diagonal (90,60)-(120,105), the dark yellow window, and places it in the array W. This will save drawing and painting another three, we can **PUT** them instead.

Lines 80, 90 and 100 places three dark yellow windows in three different locations on the house. You will notice the small difference between the **DRAW** and **PAINT** and the **PUT** commands when they are executed, with the former, the border is drawn and then the shape filled, with the latter, the complete shape is drawn and filled at the same time. This is because the complete graphics picture is stored in memory, and not the way in which it is first drawn.

Line 110 draws at location x=128, y=100, the outline of a magenta rectangle with a scaling factor of 1, a door.

Line 120 paints the door magenta.

Line 130 takes the door and stores it in memory with the variable array name D.

Line 140 places a door knob on the door.

Line 150 places a number 7 on the door.

Line 160 draws the outline of the roof and eaves in a dark green colour, using two diagonal movement commands.

Line 170 fills in the roof, in dark green.

Line 180 draws two garage doors using the front door of the house stored in memory as variable array D, but with a pixel overlap.

This effect creates a line between the two doors to simulate a join.

Line 190 defines, in small steps, a bush, B$.

Line 200 draws a bush at location x=225, y=170 with a medium green outline.

Line 210 draws another bush at location x=175, y=190 again with a medium green outline.

Line 220 paints the bushes in medium green.

Line 230 creates a flower called F$, with a green stem, but the petals depend on the variable P. The petals are drawn using the 'N' command, ensuring that all the short lines start from the top of the stem.

Lines 240 to 270 draw flowers at various locations in front of the house. Line 240 positions the first flower with an absolute blank move. Lines 250 to 270 position three more flowers using relative blank moves. Each line assigns the colour, P, of the petals before the flower is drawn.

Line 280 halts the program.

The DRAW command is a rough and ready means of drawing graphics, but nevertheless a very useful one.

In the next chapter I shall look at another simple means of drawing pictures, one that plots only one point or pixel at a time, the **PSET** and **PRESET** graphics commands.

All that remains now is to apply what we have learnt in this chapter to our ongoing problem.

****

Amend the demonstration program to use the **DRAW** command to draw the background screen.

****

The first change to make will be to line 4210 in the 'REM draw background' subroutine, to change it to:

```
4210 COLOR,4,4:SCREEN1
```

The remaining lines, which all contain **LOCATE** and **PRINT** statements using the strings created in the **'REM create strings'** subroutine, will have to be changed to **DRAW** commands, with the various strings that draw the boxes redefined within the subroutine as DRAW strings.

For example, a step, which was two characters long by one character high, can now be:

```
ST$="R12U8L12D8"
```

These ST$s must now be positioned with the DRAW command:

```
DRAW"BMx,yC15XST$;"
```

You can, of course, change the colour of each step to suit yourself by changing the C code. The x and y codes will have to be calculated from the locations in the **'REM draw background'** routine for each step, not forgetting that all the steps except the top one will be made up from more than one basic **ST$** step.

The trampoline can again be made up from these basic ST$'s, but the legs will have to redefined as **LG$**'s:

```
LG$="U16R6D16L6"
```

But they can also be used for the high jump, by building one on top of another.

The rope will still have to be the block circle, **CHR$(193)**.

You could, of course, make each of the three parts that make up the display from three separate **DRAW** commands, drawing out each piece in full; this is up to you.

As mentioned earlier the **DRAW** command can be used to create movement, and an interesting short program follows, which will give you plenty of scope for experimentation:

```
10 COLOR,15,15:SCREEN1
20 BO$="U10R10D10L10":S=1
30 X=1:Y=5:LOCATE50,10:PRINT"Colour code";S
40 DRAW"BM10,10C=S;XB0$;"
50 DRAW"BM+=X;,=Y;C=S;XBO$;"
60 X=X+1:IFX>60THENCLS:S=S+1:IFS>14THENS=0
   ELSES=S:GOTO30
70 GOTO50
```

This program draws coloured empty boxes down the screen in a curve, disappearing off the right hand side, and then waits until X>60, whereupon it clears the screen and starts all over again, but this time with a different coloured box. When all the colours have been used, the colours start at black again. The colour code is indicated at the top of the screen.

Interesting effects can be achieved by blanking out parts of the boxes by including line 55:

```
55 DRAW"BM+=X;,=Y;C0XBO$;"
```

which draws the box in transparent.

Try experimenting by changing the values of X and Y, and the X increment.

Even more effects can be achieved by changing the screen mode to SCREEN2, which will perhaps enable a better understanding of the low resolution mode. Remember each pixel, when located, fills in four pixels, not one, as in the high resolution mode. All the locations are the same though, including those used with the **DRAW** command, for example U, D, L and R.

Finally, remove the '+' sign from the X parameter in lines 50 and 55, and you will see what I mean be simulating movement with the DRAW command, the little rectangle should move right across the screen in 14 glorious colours, leaving its 'gun' behind. Or change the value of Y in line 30 to a bigger number, this way the gun will not be left behind and the whole rectangle will move across the screen.

I hope you find this experiment interesting and you can see that MSX BASIC is quite powerful.

CHAPTER SIX
Pixel Set

PSET, PRESET, POINT.

Block graphics use a 6 x 8 pixel area of the screen to draw with, as does text. Sprites use an 8 x 8, or a 16 x 16 area. **DRAW** fills in a pixel at a time to create lines in any direction, and therefore can use an area of any size of the screen you want. The graphic command **PSET** only uses an area covered by one pixel in **SCREEN 1** mode, or a rectangular group of pixels in **SCREEN 2** mode. This means, of course, that lines of any length can be drawn, of any thickness, and therefore, any size area of the screen can be used.

Depending on the instruction used to draw the line of pixels, the line can be made to bend or turn in particular directions, which means the command could be very useful for drawing graphs.

PSET, short for Pixel SET, colours in the pixel at an assigned location on the screen in the foreground colour, if no particular colour is specified, or in a specified colour if set within the command parameters.

PRESET, short for Pixel RESET, on the other hand, changes the pixel at the assigned location to the background colour, or to another colour if specified.

POINT, reads the colour of a pixel at an assigned location, and returns this colour code number as a numeric variable, or it can be used directly in a **PRINT** statement.

For example, in the following short demonstration program, using **PSET**,a pixel is randomly set to a particular colour in the centre of the screen. It is then read with **POINT**, and changed using **PRESET**, whereupon it is read again, with all the information printed out at the top of the screen.

```
10 COLOR1,7,7:SCREEN1
20 C1=INT(RND(-TIME)*15):C2=INT(RND(-TIME)
   *15):IFC1=0ORC1=7ORC2=0ORC2=7ORC1=C2GOTO20
30 PSET(128,96),C1
40 GOSUB100:GOSUB200
50 PRESET(128,96),C2
60 GOSUB100
70 K$=INKEY$:IFK$=""GOTO70ELSEGOSUB200:GOTO20
80 END
100 CODE=POINT(128,96):LOCATE25,10:PRINTCODE:
    FORD=1TO1000:NEXT
110 RETURN
200 LOCATE25,10:COLOR7:PRINT"<rg>P<rg>P
    <rg>P":COLOR1
210 RETURN
```

To get the program to repeat just press any key, line 70 does this for you.

A single pixel is such a small area that it is quite difficult to see on a normal television set. I suggest therefore that you change line 10 to:

```
10 COLOR1,7,7:SCREEN2
```

This will select the low resolution screen mode, SCREEN 2, and will set more pixels, in the shape of a rectangle, to the selected colour, the assigned one, (128,96), as the top left hand one of the group.

The PRESET command will still read this pixel, so try changing line 100 to:

```
100 CODE=POINT(129,96):LOCATE25,10:PRINTCODE:
    FORD=1TO1000:NEXT
```

That pixel, at X=129, Y=96, being in SCREEN 2, is still in the assigned colour and will now be returned with the same code as before.

Try changing the location used in line 100 to (129,97), is the same code still returned? But change the location to (128,95) and you will get 7's, the colour code for cyan, the background colour, all the time, as (128,95) is outside the area covered by the group of pixels making up the enlarged SCREEN 2 dot. This is an important point to remember for later experiments. Now try changing line 50 to:

```
50 PRESET(128,96)
```

that is, leaving out the C2 colour code change. You will find now, that each time, the colour is RESET to the background colour cyan, 7.

To double check this you could change line 10 to:

```
10 COLOR1,15,7:SCREEN1
```

But, of course, line 20 will not now error trap for the unwanted colours, those that will not show up because they are either transparent, 0, or the same colour as the background, but you will find that you get a useful rectangle of border colour, still cyan, around the displayed colour codes at the top of the screen.

This demonstrates what PRESET does if a colour is not assigned.

Finally, change line 30 to:

```
30 PSET(128,96)
```

You will now find that the pixels are always in the foreground colour, in this case black, code 1. Change line 10 to another foreground colour just to prove it, but this also means you will have to change the colour at the end of line 200 to the new colour as well!

Hopefully this short program has indicated how PSET, PRESET and POINT work, now let's go on to some movement and animation with these commands.

Type in the following short program:

```
10 COLOR12,15,7:SCREEN1
20 FORX=21TO180:PSET(X,11):NEXT
100 GOTO100
```

This program, when **RUN**, will draw a dark green straight line across the top area of the screen; X is incremented by 1 along Y=11 for 160 pixels.

Change line 20 to:

```
20 FORX=21TO180:PSET(X,11):PSET(X,170):NEXT
```

We will now get two lines drawn horizontally across the screen 160 pixels apart.

Now add:

```
30 Y=11:FORX=21TO180:PSET(X,Y):Y=Y+1:NEXT
```

which will draw a diagonal down between the two horizontal lines.

```
40 FORY=170TO11STEP-1:PSET(180,Y):PSET(20,Y):
   NEXT
```

will draw two lines up the screen 160 pixels apart, to join the two horizontal lines to complete the RECTANGLE, which tends to indicate that the pixel itself is indeed a rectangle shape, longer in the horizontal axis than the vertical.

Finally add line 50:

```
50 Y=11:FORX=180TO21STEP-1:PSET(X,Y):Y=Y+1:
   NEXT
```

which will complete the 'flag' with a second diagonal.

Now change the screen to the low resolution mode, **SCREEN 2**, and notice the thicker lines drawn. The diagonals are of course very chunky, hence the name for this screen mode, low resolution. The chunkiness can be somewhat reduced by changing the above program, that is change line 50 to:

```
50 Y=11:FORX=180TO21STEP-4:PSET(X,Y):Y=Y+4:
   NEXT
```

Now **RUN** the program again and compare the two diagonals. In line 50 allowance has been made for the fact that each pixel set in **SCREEN 2** sets a group of pixels, with the assigned pixel the top left hand one.

Now only every fourth pixel is assigned, which produces a higher resolution line.

DOTTED LINES

In addition to continuous lines, PSET can also draw dotted ones. Change line 30 to:

```
30 Y=11:FORX=21TO180STEP8:PSET(X,Y):Y=Y+8:
   NEXT
```

Now the other diagonal will be made up from a thick dotted line.

The FOR....NEXT loop steps eight pixels each time through the loop, and so only sets every other group of pixels.

Change line 40 to:

```
40 FORY=170TO11STEP-6:PSET(180,Y):
   PSET(20,Y):NEXT
```

this line will give two unevenly spaced dotted lines.

Change line 20 to:

```
20 FORX=21TO180STEP2:PSET(X,11):
   PSET(X,170):NEXT
```

will have no effect at all, do you know why? We are in SCREEN mode 2, remember!

You should remember by this stage of the book!

Now revert to SCREEN mode 1 and notice the kinds of dotted lines we have. Line 20 now has an effect on the display, were you right?

COLOURED LINES

We can also program for the lines to be self-coloured instead of being automatically in the foreground colour. To do this we must assign the colour in the PSET, (or PRESET), command, for example line 50:

```
50 Y=11:FORX=180TO21STEP-4:PSET(X,Y),1:
   Y=Y+4:NEXT
```

will draw a diagonal in black, and:

```
30 C=1:Y=11:FORX=21TO180STEP8:PSET(X,Y),C:
   Y=Y+8:C=C+1:IFC>15THENC=1
35 NEXT
```

will produce coloured dots every eight pixels, each one a different colour until all the colours have been used, and will then start the run of colours again. Occasionally a dot will be missing, do you know why?

Or again:

```
30 Y=11:FORX=21TO180STEP8:PSET(X,Y),
   INT(RND(-TIME)*15):Y=Y+8
35 NEXT
```

will produce randomly coloured dots in the diagonal.

Change to SCREEN 2 in line 10 for a more obvious result.

MOVING PIXELS AND COLLISIONS

Movement can be achieved using the SET commands, by putting them in a FOR....NEXT loop, for example:

```
10 COLOR6,7,7:SCREEN1
20 PSET(200,20),15
30 C=1:FORX=20TO256:PSET(X,20),C
40 IFPOINT(200,20)=1THENLOCATE185,20:
   PRINT"BANG":C=0:ELSEPRESET(X,20):NEXT
50 PRESET(X,20)
100 GOTO100
```

Line 10 declares the text colour to be dark red for the 'BANG' in line 40.

Line 20 positions a target, a white dot, at location X=200, Y=20.

Line 30 starts a black dot moving across the screen.

Line 40 continuously tests the white dot position to see if the black dot has arrived, if it has, then the pixel will have been changed from white to black. If it hasn't, then the PRSET command changes the previous black dot position to background colour. When the black dot arrives the IF....THEN statement tells the computer to PRINT 'BANG' at location X=185, Y=20.

Line 50 resets the black dot to the background colour after the collision has taken place.

We have achieved movement and collision detection, though the object used is not quite as big as a sprite!

Changing line 10 to SCREEN 2, will allow a bigger dot to be on the move, and is therefore more noticeable, but you will have to change the PRINT location in line 40 to LOCATE165,20 to get all the large letters in one line on the screen, but of course the rest of your display will have to be in the low resolution mode too!

A slightly larger moving ghostly block can be made to move by nesting two FOR....NEXT loops, change lines 30 and 40 to:

```
30 C=1:FORX=20TO256:FORY=20TO24:PSET(X,Y),C:
   NEXT
40 IFPOINT(200,20)=1THENLOCATE185,20:
   PRINT"BANG":C=0:ELSEFORY=20TO24:
   PRESET(X,Y):NEXT:NEXT
```

The command **POINT** is a very useful command for detecting POINTs of contact between two pieces of screen display, when either both are on the move or just one of them.

For example, amending the previous program we can get the black dot in its single pixel version to be randomly ejected from a 'gun' at the left hand side of the screen, towards a 'castle' at the right hand side of the screen. When a hit is detected an explosion, 'BANG', can occur.

Here is my amended program:

```
10 COLOR6,7,7:SCREEN1
20 T=15:GOSUB300:Y=INT(RND(-TIME)*20)+25
30 S=1:GOSUB200
40 FORX=48TO256:PSET(X,Y+7),6
50 IFPOINT(210,40)=6THENLOCATE200,38:
   PRINT"BANG":PRESET(210,40):
   GOTO80:ELSEPRESET(X,Y+7):NEXT
60 S=7:GOSUB200
70 GOTO20
80 PRESET(X,Y+7):T=7:GOSUB300
90 K$=INKEY$:IFK$=""GOTO90ELSE10
100 END
200 DRAW"BM20,=Y-20;C=S;D15R15U15L15BM+15,
    5R10D4L10"
210 RETURN
300 DRAW"BM200,40C=T;D10R20U20L5U5L10D5L5D10"
    PSET(210,40),4
310 RETURN
```

Line 10 colours the screen, choosing dark red as the foreground colour.

Line 20 assigns the colour white, code 15, to the variable T, then going to the subroutine at 300, draws a white 'castle' at X=200, Y=40, and places a dark blue dot in its middle, the target.

Line 30 assigns the colour black, code 1, to the variable S, then going to the subroutine at 200 draws a black 'gun' at X=20, Y=Y–20.

Line 40 starts to move a dark red dot, a red hot shell, across the screen starting at X=48. When you have studied the two chapters on sound why not give it some noise as well!

Line 50 checks to see if the colour of the pixel at X=210, Y=40 is dark red yet; until the 'gun' dot hits that exact location it is still dark blue. If the colour is dark red, colour code 6, then 'BANG' is printed at location X=200, Y=38 and the dot at location X=210, Y=40 is reset to the background colour. Then via line 80 the moving dot is reset and the castle is redrawn in background colour, code 7, and of course disappears. If the 'shell' has not arrived, then the last position of the 'shell', the red dot, is reset to background colour.

At line 60, once the red dot has moved across the screen and has not collided with the blue target dot inside the castle, the 'gun' is redrawn in background colour, code 7, and so disappears ready to draw another one in a new position.

Line 70 sends the computer back to the beginning of the program to redraw the gun in a new position and fire another shell, providing the shell did not hit the target. The castle is also redrawn to 'mend' any holes that the shell might have made.

Line 80 wipes out the castle on a successful hit.

Line 90 awaits a key press to start the program again.

DRAWING GRAPHS

Movement can be used in other ways, for example in drawing graphs. The following simple program draws a curve based on the square root of a number:

```
10 COLOR1,7,7:SCREEN1
20 A=1:X=20
30 PSET(X,190-SQR(A)*10),1:X=X+1:A=A+1
40 IFA=200GOTO50ELSE30
50 GOTO50
```

Sine waves can also be drawn, and similarly cosine and tangent waves, it all depends on the formula you use for incrementing the Y axis as the program moves through the X axis.

For example, the following **PSET** command will draw sine waves depending on the value of three variables P, H and N:

```
PSET(X,P+H*SIN(X/N))
```

The formula P+"H*SIN(X/N) calculates the position for the dot on the Y axis, where:
P = the dots starting position on the Y axis,
H = the maximum height or amplitude of the curve or wave drawn,
N = the approximate number of cycles drawn.

The variable X will have to be steadily incremented to move the dots across the screen, and is used in the formula for calculating the number of cycles drawn, so that the dot's Y position, it's particular height at that point, is at a position to allow the necessary number of cycles.

This next program allows sine waves to be drawn on a pair of axes:

```
10 COLOR15,7,7:SCREEN0:SCREEN0,0
20 CLS:LOCATE3,22:PRINT"Waveshape
   number";:INPUTN$
30 IFASC(N$)<49ORASC(N$)>57GOTO20
   ELSEN=VAL(N$)
40 CLS:LOCATE3,22:PRINT"1 wave or multiple
   waves, 1 or N";:INPUTN$
50 IFASC(N$)=49ORASC(N$)=78GOTO60ELSE40
60 IFASC(N$)=78THENN=1/NELSEN=N
70 GOSUB200
80 LOCATE50,180:PRINT"Again, Y or N?":
```

```
   K$=INKEY$:IFK$<>"Y"ORK$<>"N"GOTO80
90 IFK$="Y"GOTO10ELSEEND
100 END
200 SCREEN1:DRAW"BM38,10C1D140R165"
210 FORX=40TO196
220 PSET(X,70+50*SIN(X/N))
230 NEXT
240 RETURN
```

Swapping between screens in a program can sometimes cause trouble on the Spectravideo, in that you do not get the screen you want, especially if you are using INPUT, which as mentioned previously, will not work on a high or low resolution screen. It is a good idea therefore at line 10, if INPUT is to be used, and the function keys are not to be displayed, to change to `SCREEN0,0` to remove the function windows. As the program, having already drawn a sine wave, can come back to line 10 to do it all again, this is what I have done here.

Line 20 prompts the user to type in a wave shape number, the bigger the number the fewer the number of cycles. In fact, in this example of the program, the number 25 gives one complete cycle, and anything over 1000 is liable to give a straight line.

Line 30 checks that the first number typed in, you can type in any number except 0, is in fact between 1 and 9. To carry out the division in the formula in line 220 we naturally need a number, but here I have used a string. This allows me to check that 0 has not been typed in, the ASCII codes allowed are between 48 and 57, decimal numbers 1 and 9, and also that a letter hasn't been accidently or deliberately typed in, their ASCII codes are greater than 57, the alphabet starts at 65. This, of course, is a subtle type of error trapping.

Line 40 then asks if a single wave is to be drawn or multiple ones.

Line 50 checks, in the same manner as line 30, that only the required 1 or N is typed in, and sends the computer to ask for the information again if they aren't. Don't forget that only a capital N has been allowed for. If you want to allow for a lower case m, then extra ASCII codes must be put into the statement.

Line 60 checks the value of N$ again. If N$="78", the ASCII code value of capital N, then it turns the inputted number into its reciprocal, 1/N, which enables the computer to draw multiple waves instead of just the one. Three phase wave shapes are normally drawn, though it is possible to get two phase with some numbers. If the inputted number is 1 then only one wave is drawn, N=N.

Line 70 sends the computer to the subroutine for drawing the wave shape.

Line 80 asks the user if another display is required, and ensures that only capital Y or capital N is typed in.

Line 90 deals with both Y and N. Y sends the computer back to the beginning of the program, N ends the program.

Line 100 is a safety END program line, not strictly necessary here.

Line 200 changes the display to a high resolution screen, and draws the Y and X axes for the graph. Try changing this **DRAW** command to a **PSET** command, you will see it's a lot easier using the macro graphics language, **DRAW**.

Line 210 starts the **FOR....NEXT** loop for incrementing the dot across the X axis.

Line 220 draws the curve or wave shape, at a particular height and position, depending on the value of N.

Line 230 completes the **FOR....NEXT** loop.

Line 240 returns the computer to the main program.

Lines 200 to 240 could of course all be written on one program line.

In this chapter we won't be attempting to change any of the ongoing problem program. **PSET**, **PRESET** could be used to draw the display, but it is much easier with **DRAW**. **POINT** could be used to detect whether the gymnast has cleared the high jump, but then he would have to be drawn using a graphics command, and sprites are more convenient for this. Consider then this small respite as a half term holiday, seeing as we are approximately half way through the book.

In the next chapter we shall be looking at the final commands of the graphics language on the Spectravideo and in MSX BASIC, **LINE** and **CIRCLE**, two very versatile commands.

CHAPTER SEVEN
The Circle Line

LINE, CIRCLE

In this final chapter on simple graphics I shall be looking at the two commands LINE and CIRCLE.

DRAWING LINES

But before I start it will be a useful exercise I think to compare the four means of drawing a line in MSX BASIC on the Spectravideo. These are:

1. PRINTing a graphics character,
2. DRAWing a line,
3. PSETing pixels, and,
4. LINE drawing.

PRINTing, of course, relies on the fact that there are block graphics characters on the keyboard to PRINT with, and the ones that will draw a one pixel thick line in a horizontal direction are <lg>L, and <lg>O, and in a vertical direction are <rg>L and <rg>O. As each character is printed the computer must be told to step on the correct number of pixel locations, because PRINTing in SCREEN 1 or SCREEN 2 requires the use of the LOCATE statement, and stepping on requires the use of FOR....NEXT loop as well. The line is coloured in the present specified text or foreground colour, unless this is changed just prior to the PRINT statement.

DRAWing only requires the start location, usually with a blank move, or a move relative direct from a present position, and the information necessary to draw the line a given number of pixels. Colouring uses either the present foreground colour, or one specified in the DRAW command.

PSETting sets one pixel at a time, but again requires a FOR....NEXT loop to draw a continuous line of pixels, but does tend to be rather slow, as a separate action on the computer's part is required for each pixel to be set.

The graphics command LINE is designed for the job, but will only draw lines between the specified locations X1,Y1 and X2,Y2 in the command:

```
LINE(X1,Y1)-(X2,Y2)
```

It automatically LOCATEs the first location, a blank move, and then draws a line to the second one, but if the line drawn isn't straight through a continuous line of pixels then the line may be staggered.

To demonstrate these facilities type in the following short demonstration program:

```
10 COLOR1,7,7:SCREEN1
20 LOCATE20,10:PRINT"PRINT":FORX=61TO210
   STEP6:LOCATEX,10:PRINT"<lg>L":
   NEXT:GOSUB200
30 LOCATE20,50:PRINT"DRAW":DRAW"BM61,
   50R150":GOSUB200
40 LOCATE20,90:PRINT"PSET":FORX=61TO210:
   PSET(X,90):NEXT:GOSUB200
50 LOCATE20,130:PRINT"LINE":
   LINE(61,130)-(210,130)
100 GOTO100
200 FORD=1TO800:NEXT:RETURN
```

You should get four single pixel width lines across the screen together with a description of the command or statement used for each one.

PRINT you will notice has its line drawn at the bottom of the word, as we used <lg>L, to get it to the top like all the others you should use <lg>O.

DRAW, **PSET** and **LINE** have the line at the top of the word, indicating that the position for the pixel located by the location codes is the top left hand corner of the group. This is more obvious if you **RUN** this short program in **SCREEN 2**.

DOTTED LINES

To create a dotted line effect with these four methods amend the program as follows:

```
20 LOCATE20,10:PRINT"PRINT":FORX=61TO210
   STEP12:LOCATEX,10:PRINT"<lg>L":NEXT:
   GOSUB200
```

All we need to do here to produce dotted lines is to increase the **STEP** variable, any number over 6 will produce a dashed line.

```
30 LOCATE20,50:PRINT"DRAW":DRAW"BM61,50":
   FORS=1TO12:DRAW"BM+6,0R6":
   NEXT:GOSUB200
```

Here the DRAW draws a line of 6 pixels, R6, after moving on from the last location 6 pixels, BM+6, twelve times.

```
40 LOCATE20,90:PRINT"PSET":FORX=61TO210STEP6:
   FORS=1TO6:PSET(X,90):X=X+1:NEXT:
   NEXT:GOSUB200
```

In this line using **PSET**, again the command is put into a **FOR....NEXT** loop which is programmed to set 6 pixels at a time, before stepping on another 6 pixels, STEP 6.

```
50 LOCATE20,130:PRINT"LINE":X=61:FORS=1TO13:
   LINE(X,130)-(X+6,130):X=X+12:NEXT
```

Here the value 61 is first assigned to the variable X, and the LINE command is put into another FOR....NEXT loop to draw a short line of 6 pixels. Then the variable X is incremented by 12, which has the effect of starting the next short line, or dash, at a point 6 pixels on from the previous one.

COLOURED LINES

Obviously the lines drawn can be coloured as well, the PRINT requiring a COLOR change before and after the line drawing command. The DRAW requires its colour code inside the DRAW command, which means that each dash could also be a different colour, if required. The PSET command requires any colour command to be after the location codes, and again could be programmed to have different colours for each dash.

For example:

```
30 LOCATE20,50:PRINT"DRAW":P=1:DRAW"BM61,50":
   FORS=1TO12:DRAW"BM+6,0C=P;R6":P=P+1:
   IFP=7THENP=8
35 NEXT:GOSUB200
```

will draw each dash in a different colour, and also allow for the case when the colour is cyan, code 7, by changing it to code 8, with IFP=7THENP=8. Notice that the NEXT:GOSUB200 has been moved onto a new line, line 35, otherwise the program will stop at line 30 after only one dash.

The LINE command also requires a colour code after the pair of location codes:

```
LINE(X1,Y1)-(X2,Y2),C
```

where C equals the colour code.

Now DELETE the short program, lines 20 to 50 we've been experimenting with, and type in the following line:

```
20 LOCATE20,20:PRINT"LINE":LINE(61,20)-
   (210,20),4
```

which will draw a long blue line at the top of the screen. If the colour code is left off then the line will be drawn in the assigned foreground colour. Delete the ,4 from line 20, and prove it!

LINE is a very versatile graphics command, it will also draw rectangular shapes, squares as well.

Amend line 20 to the following:

```
20 LINE(31,20)-(90,60),4,B
```

When RUN this short program will draw a blue rectangle, exciting isn't it?

Compare this line to the DRAW command in the new line 30:

```
30 DRAW"BM31,80C4R60D40L60U40"
```

which also draws a blue rectangle of the same size.

But here is the big difference, to fill in the shape with colour, the **DRAW** command needs a **PAINT** command:

```
35 PAINT(33,82),4
```

and when RUN the filling takes a little time, and if you don't get the colour codes right or you make a mistake with the locations the rest of the screen will fill with colour instead.

But with the **LINE** command all you have to do is add the letter F to the B in line 20!

```
20 LINE(31,20)-(90,60),4,BF
```

Now when **RUN** compare the two rectangles and the speed with which they fill!

If you only want the shape to fill with the assigned foreground colour then you must still put in the two commas which would have been either side of the colour code:

```
20 LINE(31,20)-(90,60),,BF
```

If you only put in the one comma, as you might expect would be permissible, then the computer will try to draw the lines in colour code BF, and will try for ever, as the program will not crash, but just 'hang' indefinitely:

```
20 LINE(31,20)-(90,60),BF
```

If, somewhere in your program you have assigned a value to a variable BF, then you will get a diagonal line instead of a filled shape.

```
20 LINE(31,20)-(90,60)BF
```

will crash with a syntax error.

```
20 LINE(31,20)-(90,60),,BF
```

will produce a black square in our program.

It is also possible to draw on top of other shapes using the LINE command:

```
25 LINE(46,30)-(75,50),6,B
```

will draw the outline of a rectangle in the middle of the black square, and

```
25 LINE(46,30)-(75,50),6,BF
```

will fill it in in dark red.

Lines can be drawn in different directions in a continuous fashion, simply be extending the LINE command. Delete, with:

```
DELETE20-200<ENTER>
```

the unwanted lines for the next part of the experiment and notice how the extra lines are achieved by using LINE with only the second location bracket, for example:

```
20 LINE(30,20)-(70,20):LINE-(70,60)
100 GOTO100
```

will draw a first line 40 pixels long, and then a second line 40 pixels downwards at right angles.

Similarly:

```
20 LINE(30,20)-(70,20):LINE-(70,60):
   LINE-(30,60):LINE-(30,20)
```

will draw a square, in the same way that:

```
20 LINE(30,20)-(70,60),,B
```

will. But the following line will draw a five sided figure:

```
20 LINE(50,20)-(80,20):LINE-(95,35):
   LINE-(80,50):LINE-(50,50):LINE-(50,20 )
```

and will give you some idea of how a diagonal is displayed. The angle through which it has to travel will determine the straightness of the line, or whether it is staggered or not. For example change line 20 to:

```
20 LINE(50,20)-(80,20):LINE-(125,85):LINE-
   (80,50):LINE-(50,50):LINE-(50,20)
```

which will produce an oddly shaped five sided figure but will indicate the amount of stagger produced.

Filling this odd shape with colour will require the use of the PAINT command, because neither the 'B' nor the 'F' parameter could have been used in the LINE command. But the only colour that we can fill the shape with is black, because black was originally called in line 10 of our original program, we didn't delete line 10 remember! To get the shape to fill with some other colour requires each part of the outline of the shape to be given a colour code, the same colour code. We can always have shapes with different coloured sides, but we are then not allowed to fill them. The PAINT colour MUST ALWAYS be the same colour as the sides of a shape or the whole screen is liable to be filled instead!

To get our five sided shape to fill then, we must assign the same colour code to each LINE command, as follows:

```
20 LINE(50,20)-(80,20),12:LINE-(125,85),12:
   LINE-(80,50),12:LINE-(50,50),12:
   LINE-(50,20),12
```

This will now draw the shape in green, and fill it in green too.

If you don't believe me then try changing one or two of the colour codes!

By the way, it would be difficult to draw this shape with the DRAW command.

To get some idea of the resolution of diagonal lines, and as an example of the versatility of MSX graphics, change line 20 to the following, and add line 30:

```
20 X=20:Y1=96:Y2=0
30 FORDR=0TO196:LINE(X,Y1)-(X+100,Y2):
   Y2=Y2+1:NEXT
```

As this short routine is running you will be able to see the change in resolution of the diagonals as they are drawn, and to stop the display at any point, of course, you can press the **STOP** at the top of the keyboard. To restart the display press the **STOP** button once again.

To get more individual lines, and not a filled shape, change the value of the Y2 increment to a bigger number, say 5 or 10. A value of 25 produces one diagonal of each resolution.

And so to circular shapes.

CIRCLES

The graphics command **CIRCLE** is similar to the **LINE** command, in that it contains a number of parameters, but does not have the facility for automatically filling the shape with colour, the **PAINT** command would have to be used to do this.

To draw a circle change line 20 to:

```
20 CIRCLE(128,96),10
```

This will draw a small circle in the centre of the screen, but the circle would be better called an ellipse. The two numbers in the bracket are, as usual, the X and Y locations of the centre of the circle, and the first figure outside the bracket is the radius in MSX units, whatever they are!

To create a true circle then we must employ another parameter of the **CIRCLE** command, that which gives it a particular 'aspect ratio', or its shape with regard to its two axes, X and Y. Change line 20 to:

```
20 CIRCLE(128,96),10,,,,1.4/1
```

This will produce a reasonably round circle, but the horizontal and vertical areas may be straight, depending on it's size. The reason for the four

commas will be explained shortly, for the time being just make sure there are four, or you won't get a circle!

The aspect ratio is now 1.4 to 1, which means that the vertical axis is now 1.4 times more than the horizontal as regards length, and the ellipse effect has been cancelled out. One point to note is that although this works on my television, it may not on yours, it will all depend on how your television horizontal and vertical axes are set, and there's nothing I can do about that. So to produce near enough perfect circles, when you want them you will have to use this aspect ratio, but experiment with the two numbers to suit your television, and then make a note of the ratio required.

To indicate the resolution of the circles depending on how big the radius is change line 20 to:

```
20 FORR=0TO96:CIRCLE(128,96),R,5,,,1.4/1:
   NEXT
```

which will slowly display a large round blue plate. The cyan dots in it indicate the pixels that have not been filled in due to the fact that the circular shape is not truly circular, but sets two or three pixels in a straight line now and again. Now you can see one of the reasons for the extra four commas. One is to accommodate the colour code for the outline of the circle.

MAGIC CIRCLES AND EXPLOSIONS

To further illustrate the versatility of a machine using MSX BASIC, change line 20, and add line 30:

```
20 C=0:FORR=0TO96:CIRCLE(128,96),R,C,,,
   1.4/1:C=C+1:IFC>15THENC=0
30 NEXT
```

This will have a spectacular effect on the 'plate', and will also indicate how adjacent pixels, depending on their location, have an effect on each other. The effect is also useful for simulating explosions in arcade games and the like, but using much smaller loops of course.

SEPARATE CIRCLES

Now to get back to normal graphics change line 20 to:

```
20 C=1:FORR=0TO96STEP4:CIRCLE(128,96),R,C,
   ,,1.4/1:C=C+1:IFC>15ORC=7THENC=1
30 NEXT
```

This will produce different coloured circles at a four pixel separation, but you will notice that the colour still has an effect on adjacent pixels even at this distance. You will probably find that you will have to go as far as a STEP of 12 to stop this happening, but this will also depend on where the centre point of the first circle is.

You will also notice that the circle is drawn in four parts, which allows

the computer to present the next facility, and the reason for the extra three commas. We can also draw parts of a circle, and this all depends on what two numbers we put between the other three commas.

Putting 2PI in both places will not draw a circle, it confuses the computer. 2PI is the true circumference of a circle, half of this, 1PI, is therefore only half a circle, 0.5PI is a quarter, and so on. The first PI number will tell the computer where to start drawing the circle, and the second PI number where to stop. For example:

```
20 C=0:FORR=0TO96:CIRCLE(128,96),R,C,0,
   22/7,1.4/1:C=C+1:IFC>15THENC=0
30 NEXT
```

tells the computer to start at position 0 and draw as far as 22/7, 22/7 is PI, as is 3.142, you can choose your own way of doing it.

A '0' in the first PI location tells the computer to start at three o'clock. This particular line 20 will draw a RAINBOW!!

Changing the two PI numbers around will draw a rainbow reflection, to do this add lines 40 and 50:

```
40 C=0:FORR=0TO96:CIRCLE(128,96),R,C,
   22/7,0,1.4/1:C=C+1:IFC>15THENC=0
50 NEXT
```

and temporarily delete line 30, which has the effect of stopping line 20 from being executed!

Now insert line 30 again, and you will get the multi-coloured circles back again, proving that the circles are drawn from 2PI to 2PI, or 22/7 to 22/7!

Delete lines 40 and 50.

Change line 20 to:

```
20 C=0:FORR=0TO96:CIRCLE(128,96),R,C,0,11/7,
   1.4/1:C=C+1:IFC>15THENC=0
30 NEXT
```

This will have the effect of drawing only quarter circles, I prefer the 22/7 PI number for this reason, you can easily work with it, more easily than 3.142, which is only approximate anyway.

Now adjust line 20 to:

```
20 C=0:FORR=0TO96:CIRCLE(128,96),R,C,0,5.5/7,
   1.4/1:C=C+1:IFC>15THENC=0
30 NEXT
```

This will draw eight circles or small arcs, or a nice slice of birthday cake.

MORE GET AND PUT

Typing NEW<ENTER> will allow us to type in a new program without deleting all the lines of the old program.

Now type in the following few lines, I'll explain them as we go, which will also explain the full uses of GET and PUT:

```
10 COLOR1,15,7:SCREEN1:DIMR(10,13)
20 CIRCLE(126,96),90,12,,,6/1
30 PAINT(126,96),12
```

These three lines will colour up the screen, then draw a filled in dark green tall elipse.

```
40 LINE(31,171)-(220,180),2,BF:LOCATE31,140:
   PRINT"LINE"
```

will draw a medium green long rectangle at the bottom of the screen, and then print the word line on the right hand end of it.

```
50 GET(31,171)-(220,180),R
```

will place this area of the screen in memory, to be used later with the PUT command. The variable R has to be dimensioned correctly, using an array, and this is what I have done in line 10. But how did I know what numbers to use in the array?

A simple way to carry out the calculation to arrive at the two numbers in the DIM bracket is to take the difference between the X locations, and the difference between the Y locations, and then multiply these two numbers together.

Then divide the result by 15. This final answer is the multiple of the two numbers in the DIM bracket, and the easiest way to get this is to divide the final answer by 10, and then use 10 and this answer as the two numbers.

For example, I had an X difference of 190, 220-31, and a Y difference of 10, 180-171, which gives me, XxY=190x10=1900. Then divide this 1900 by 15, which gives 126.66666666667, or 127 to the nearest whole number. Divide this answer by 10, and we get 12.7, which to the nearest whole number again gives 13. Therefore I dimensioned my array as (10,13), but it could just as well have been (13,10). You could then, if you were short of memory in a program you were writing, start to reduce one or both of the DIM numbers until the program came up with the error 'ILLEGAL FUNCTION CALL', whereupon you could go back one step in your experiment.

```
60 KEYON:ONKEYGOSUB200,300,400,500,600,700
70 GOTO70
```

Here I have used the ON KEY command to tell the computer what to do when a function key is pressed. First initalise this facility with the command KEY ON, then after the ONKEYGOSUB statement, put the various subroutines that the computer must go to in numerical order,

that is KEY 1 will send it to 200, KEY 2 to 300, etc. The subroutines can be in any order, but the the computer will assign them in positional order to the function KEY numbers.

Line 70 justs keeps the program from ending when any of the subroutines have been executed.

Here are the six subroutines:

```
200 PUT(31,140),R,PSET:LOCATE31,140:
    PRINT"PSET"
300 PUT(31,110),R,PRESET:LOCATE31,110:
    PRINT"PRESET"
400 PUT(31,80),R,AND:LOCATE31,80:PRINT"AND"
500 PUT(31,50),R,OR:LOCATE31,50:PRINT"OR"
600 PUT(31,20),R,XOR:LOCATE31,20:PRINT"XOR"
700 SCREEN0:LIST:END
```

Now **RUN** the program and stand by for instructions!

Each subroutine, 200 to 600, **PUT**s the **GET** rectangle, R, in a particular position on the screen, then prints the means of colouring it on the left hand side of the rectangle. These means of colouring it all depend on the use of a particular operator at the end of the **PUT** command.

These are explained as follows:

PSET: colours the area selected by the GET command in the same colour as it was originally. Press the F1 key to prove it.

PRESET: colours the area in the inverted colour, that is the colour code arrived at by the formula 15 minus the original colour code number. Here this is 15-2=13, which gives the colour code 13, magenta. Press the F2 key and see this magenta rectangle appear.

AND: colours the area according to a binary calculation as follows. Translate the colour codes of the colour being PUT and the colour already on the screen display in the area where the new PUT area is to go, into their binary equivalents from the following table:

colour	**code**	**binary number**
transparent	0	0000
black	1	0001
medium green	2	0010
light green	3	0011
dark blue	4	0100
light blue	5	0101
dark red	6	0110
cyan	7	0111
medium red	8	1000
light red	9	1001
dark yellow	10	1010
light yellow	11	1011

dark green	12	1100
magenta	13	1101
grey	14	1110
white	15	1111

Then obtain the new code by the use of AND on the two colours present according to the following rules:

1 if two 1's, 1 AND 1=1

0 if a 1 and a 0, 1 AND 0=0, 0 AND 1=0

0 if two 0's, 0 AND 0=0.

For example we have when added together:

medium green 0010, *together with*
a white background 1111, *giving*
a result of 0010, *colour 2, or medium green again.*

Press the F3 key to find out.

We also have the medium green rectangle on top of the dark green ellipse, so

medium green 0010, *together with*
dark green 1100, *giving*
a result of 0000, *colour 0, or transparent.*

You will find that the area over the ellipse is indeed transparent, it shows the border colour through, colour code 7, or cyan! Change the border colour yourself and experiment. To do this press <SHIFT><F1> to get the listing, this is what the subroutine at line 700 is there for.

Now RUN the program again.

OR: colours the area according to the following rules:

1 OR 1=1
1 OR 0=1
0 OR 1=1
0 OR 0=0

We have:

medium green 0010, *together with*
a white background 1111, *giving*
a result of 1111, *colour 15, or white.*

Press the F4 key to find out.

We also have the rectangle on top of the dark green ellipse, so

medium green 0010, *together with*
dark green 1100, *giving*
a result of 1110, *colour 14, or grey.*

The area inside the ellipse is indeed grey. You can experiment by changing the colour of the background or the ellipse.

XOR: colours the area according to the following rules:

1 XOR 1=0
1 XOR 0=1
0 XOR 1=1
0 XOR 0=0

we have:

medium green 0010, *together with*
a white background 1111, *giving*
a result of 1101, *colour 13, or magenta.*

Press the F5 key to find out. We also have the rectangle on top of the dark green ellipse, so

medium green 0010, *together with*
dark green 1100, *giving*
a result of 1110, *colour 14, or grey.*

Again you can experiment to prove the rules.

You will notice that the new areas are not exactly in line with the old in the ellipse, this is due to the fact that the colours creep once you have been removing and replacing bits with PUT. You will also notice that the letters LINE are stored in the array as well, but don't necessarily reappear, it all depends on the colour used for the foreground or text, sometimes you just get a blotch of colour instead.

A big area for you to experiment with there!

Finally, if you do experiment and find that when you press <SHIFT><F1> to stop the program and get the listing, nothing appears but a blank screen, press <SHIFT><F1> again, this should colour the text the right colour for you to be able to see it. If you are not quite sure then type KEYLIST<ENTER> and you'll get a list of all the programmed function keys, number 6 is COLOR 15,4,5.

PROBLEM TIME

Applying your new found knowledge to the ongoing gymnastic display problem program should be very easy, and I'll not give you any clues on what to do.

But in the chapter on DRAW I drew a screen display of a house, garage, bushes and flowers. So let's attack a new problem.

****

Rewrite the DRAW screen display in chapter five using LINE, CIRCLE, GET and PUT, making the windows all different colours.

****

Rewriting this program should not prove too difficult either.

I would suggest that you experiment with the PUT operators, PRESET, AND, OR and XOR to get the windows different colours. You could also make the bushes from small circles, and maybe also put an umbrella in the garden with the CIRCLE and LINE,BF commands.

The next two chapters will deal with the SOUND facilities of MSX BASIC and the Spectravideo.

CHAPTER EIGHT
Play Strings

PLAY

MSX BASIC and the Spectravideo offer a marvellous range of music making and sound effect facilities, and with it own built-in sound synthesizer, it does not have much competition from other micro computers, (at the time of writing).

PLAY is the command used to produce music, and **SOUND** the one used to produce sound effects, although this command can also produce music, it does tend to do it in a more complicated manner.

Sounds are vibrations of the air that are picked up by the ear, and which, via the brain, are interpreted into the sounds that are all around us. Musical sounds are purely those sounds that to man are pleasant and worth listening too. Different races of people throughout the world have different ideas of what is a musical sound and what is not, but in the main there are certain frequencies, or speeds of air vibration, that are accepted as being 'right', and to this end have been laid down as a set of rules.

Craftsmen spend their lives touring their localities 'tuning' instruments of various sorts to these particular frequencies, so that they will be 'in tune' with all the others. But each separate instrument has different harmonics, extra frequencies based on the main one being played, that alter the way in which we hear a particular sound. To achieve this effect we have to use the **SOUND** command in MSX BASIC, this command will be dealt with in the next chapter.

The **PLAY** command can only produce the main or fundamental frequency or **NOTE**, but it can be disguised, as we shall see later. The sound you hear has to come from your TV sound system, fed to the TV from the computer by the video lead, which itself introduces various extra harmonics, which are then added to the note being played. That is why different makes of TV produce slightly different sounds for the same music.

NOTE PLAY

The command **PLAY** can produce the seven notes that go to make this set of musical rules, and these seven notes are named as follows:

A B C D E F G

Each of these seven letters has a particular frequency associated with it. The letters are not normally arranged in this order, but:

C D E F G A B

which go to make up the complete set of seven notes, called a scale or OCTAVE.

To complicate matters more all the frequencies can be doubled to produce the same NOTE but at a higher frequency, and so, as our ear will vibrate faster on receiving this sound, we hear a different sound, a 'higher' sound or NOTE. We say then that the note is in a higher OCTAVE. The Spectravideo can produce eight different OCTAVES, 1 to 8, 1 being the lowest frequency set and 8 the highest. This means that we have a range of 7x8=56 notes available to PLAY, from OCTAVE 1, NOTE C or O1C to OCTAVE 8, NOTE B or O8B.

Let us experiment now with the computer to produce any one of the seven available notes in OCTAVE 4, the octave that the computer powers up in when you first switch on, called the 'default' octave.

Type in the following short program:

```
10 COLOR1,7,7:SCREEN0,0
20 LOCATE6,10:PRINT"CAPS LOCK ON"
30 LOCATE6,12:PRINT"note please";
40 INPUTN$:IFASC(N$)<65ORASC(N$)>71GOTO10
50 IFN$="H"THENSCREEN0,1:LIST
60 PLAYN$
70 FORD=1TO450*LEN(N$):NEXT:GOTO10
```

Lines 10 to 50 should not really need any explanation at this stage in the book, but music might be your reason for buying the Spectravideo,and you may have opened the book at this chapter first, then turn to chapter eleven and look up any statements or commands in these lines that you do not understand.

Line 60 tells the computer to PLAY the note that is contained in N$, and this will be either C, D, E, F, G, A, or B, line 40 ensures this.

Line 50 tells the computer to END the program if key 'H' is pressed.

Line 70 ensures that you do not get the prompt screen again until the note, or notes, have finished playing.

When you RUN the program you will be asked for the note you wish to hear, press that key and then the ENTER key, and listen.

Experiment with all the seven notes, running up the scale and down again, and compare them, but don't forget to turn the volume up on your TV set, don't forget to press the CAPS LOCK key, and check that the light is on!

Now when you have heard enough of the single notes, take the experiment one step further, and press more than one key per screen prompt, for example:

```
CDEFGAB<ENTER>
```

You will get the first seven notes of the scale C Major, or the notes in OCTAVE 4.

To those with an 'ear for music' something will of course be missing, the last note in the scale, the higher C, and we'll deal with that shortly.

For the moment try composing short melodies, for example:

```
CEDFEGC<ENTER>
```

Be careful not to include non-note letters in the string or the program will crash. If you do, just press F5 to RUN the program again.

How does the computer play these notes?

Each letter that you type in is placed, in the normal fashion, in a string variable, N$, which the computer reads and translates into the required frequency or frequencies. The computer plays PLAY strings, in exactly the same way that it draws DRAW strings.

So in command mode you can type:

```
PLAY"CDEFGAB"<ENTER> or
PLAY"cdefgab"<ENTER> or even
PLAY"CdEfGab"<ENTER>
```

and produce the musical notes. The reason I asked you to type in capital letters was to save complicated `IF....THEN` statements.

Experiment now in command mode typing in your own `PLAY` commands.

SHARPS AND FLATS

You can also get the computer to play sharpened and flattened notes, notes and which are slightly altered from the original, that are used to create different scales. To do this follow the note letter with a <SHIFT>3 or a <SHIFT>0 or '+', for the sharpened note, a '–' produces a flattened note.

But this means we will have to amend our program a little and remove the error trapping or safety precautions from it.

Our amended demonstration program will now look like this:

```
10 COLOR1,7,7:SCREEN0,0
20 LOCATE3,20:PRINT"N$ = "N$:LOCATE6,12:
   PRINT"notes please";:INPUTN$
30 IFN$="H"ORN$="h"THEN SCREEN0,1:LIST
40 PLAYN$:FORD=1TO150*LEN(N$):NEXT:GOTO10
```

You must be extra careful from now on not to type in any wrong notes or symbols, or any in the wrong order, if you do the program will 'crash'. Remember just press F5 to RUN it again.

We could error trap this program completely, but it would take some tedious `IF....THEN`ing.

Also, notice the use of the extra `LOCATE` and `PRINT` in line 20, this

will show you the state of N$. If you want to hear the same sequence again all you have to do is press <ENTER>, any other key, followed by <ENTER>, will immediately change the state of N$.

To show you quickly what a sharpened or flattened note produces, RUN the amended program and type in:

```
C+D-<ENTER>
```

and you will only get one long note, this is because C+ = D–, or C sharp = D flat. You can now experiment and find out what all the other sharpened and flattened notes are equivalent to!

You will perhaps also notice now that all the notes last for the same amount of time, either in my program or in command mode.

This is due to the fact that the computer powers up with a certain default note length, L8, a crochet's worth in musical terms. A crochet is a filled in black circle with a line attached to it, but see table 1 for a fuller explanation of all the notes and their lengths that you can have in musical scores.

LONG AND SHORT NOTES

To get longer notes with my program, or in command mode, type the same letter twice for a minim, or four times for a semibreve, three times will give you a dotted minim. A dot after a note increases the length by a half again, and this even works in the program string, try it, and type in a dotted crochet. A 'C.' should do it.

You can now have any length of note you want greater than a crochet.

This also means that you can't have two notes the same together, two C's produce a double length note. But you can type in a 'R' between the notes, this tells the computer to Rest before playing the next note, but the Rest will be the same length as the single note, called R8 in MSX language.

So how do we get notes and rests shorter than a crochet?

The length of a note on the computer is controlled by the letter 'L' or 'l', but I would advise the use of the capital letter only, to avoid confusion with the number 1, not for the computer, but for you!

Each note therefore can be programmed to last a given time by typing in the letter l and a number. The length command, L, will be executed on each note in a string until changed by another L command.

The command 'L' has the values 1 to 64, with 1 having the longest value, and 64 the shortest, see the table of note values. So try:

```
L64CL32DL16EL8FL4GL2AL1B<ENTER>
```

to get a feel of all the values. Of course, there is nothing to stop you

using any number between 1 and 64 for the length of your note, it's up to you, you're writing the music!

RHYTHM AND SPEED

Tempo is another way of changing the length of your notes, in fact, as its name implies, this command changes the whole speed of the PLAY string, and alters the length of the range of L commands to suit. For example, type in:

```
TCDEFG<ENTER>
```

This will play the notes in the normal fashion, at the default speed or tempo, T=64. Type in:

```
T64CDEFG<ENTER>
```

to prove it.

Now try other values of T, 255 is the quickest and 32 the slowest.

Careful, values less than 32, or greater than 255, are not allowed.

Now try:

```
L32CL16DL8E<ENTER>
T32L32CL16DL8E<ENTER>
T255L32CL16DL8E<ENTER>
```

and notice the difference, or even:

```
T128L32CDET255EDCT32DEC<ENTER>
T128L32CDF+T255B-CDL2DFG<ENTER>
```

So far we have only been playing in the scale of C Major with only seven notes, now let's get a little more adventurous by using the OCTAVE, O, command.

Type in:

```
O4CDEFGABO5C<ENTER>
```

which will now play the complete scale, middle C to C'. Or:

```
O5CO4BAGFEDC<ENTER>
```

which will play the scale in reverse order, or

```
O4CDEFGABO5CO4BAGFEDC<ENTER>
```

and you'll get the normal practice scale.

Placing T32 in front of this particular N$ will of course play it quite a lot slower.

We can also change the volume in a string as it is being played, by

using the command V for Volume. V has the values 1 to 15, with 15 the loudest and 1 the quietest. Try:

```
V15CV13DV11EV9FV7GV5BV3AV105C<ENTER>
```

but plug your eardrums first!

If you play this more than once by just pressing <ENTER>, then you will notice that you start in the O5 octave, showing that the PLAY commands remain as they are set until changed - this is an important point in writing computer music. Can you work out how to keep the octave playing the C Major scale, however often you play by repeated pressing of the <ENTER> key? Oh, for a clue!

SOUNDS EXTRAORDINARY

There are two other commands we can experiment with, using this short demonstration program. These are S for SHAPE and M for MODULATE. S, as its name implies, gives the notes a certain pre-programmed shape or envelope, (more about that in the next chapter,) and M makes the whole note wobble or modulate.

The command 'S' has the values 0 to 15, and the command 'M' 1 to 65535, and tends to replace any 'T' and 'V' commands we may have used.

M really has to jump in thousands to make any noticeable difference, but you will get some amazing surprises. It's up to you to experiment.

Try these for starters:

```
S0M1000CDE<ENTER>
S4M1000CDE<ENTER>
S8M1000CDE<ENTER>
S15M1000CDE<ENTER>
S15M1CDE<ENTER>
S15M12000CDE<ENTER>
S10M10000CDEFGAB<ENTER>
S9M4000CDEFGAB<ENTER>
S11M8000L64CDEFGABO5C<ENTER>
```

The combinations are almost limitless, but many of the sounds you can make are the same, at least to the human ear. I haven't tried them on the neighbour's cat, though I have been tempted at times.

The length of the note used does have some effect on the shaping and modulating of the notes, for example, try:

```
S2M4000L404CDEFGABO5CL32CO4BAGFEDC<ENTER>
```

notice the difference in the rising and falling scales produced. It is even more noticeable if you use an L of 64, the shortest note.

Notes are automatically separated when using the S and M commands, so that:

```
S1ØM4ØØØCCCC<ENTER>
```

will produce four separate notes. Similarly:

```
S1ØM4ØØBBBBBB<ENTER>
```

will produce an alarm call!

The S and the M commands can be used separately, together with the other commands, but then the default value, or the value last used, of the missing command will be used by the computer.

For example:

```
S1ØM3ØØCDEFGAB<ENTER>
```

and:

```
S1ØCDEFGAB<ENTER>
```

and:

```
M3ØØCDEFGAB<ENTER>
```

and:

```
CDEFGAB<ENTER>
```

will here all produce the same very interesting sound, notice M=300 only here.

Once having used the M and S commands while developing a program, it is always a good idea to **CSAVE** it, switch off the computer, then **CLOAD** it back in again, and test all your **PLAY** commands in the program. Gremlins are everywhere. Typing in **M65535** in my program should get the M command back to normal, as does **S8**, but you will have to adjust the volume with V8 to suit. So:

```
M65535S8V8<ENTER>
```

should return our program to normal.

```
PLAY"M65535S8V8"
```

will hopefully do it under normal circumstances without my demonstration program.

If you want to save the demonstration, I should do it now, as we are now going to **PLAY** with different strings.

STRINGING IT ALTOGETHER

As we saw with the DRAW command, strings can be joined together by 'adding' them, for example, type in, in command mode:

```
M1$="03CDE"<ENTER>
M2$="05EDC"<ENTER>
M3$="06ABC"<ENTER>
```

Then:

```
PLAYM1$+M2$<ENTER>
PLAYM2$+M1$<ENTER>
PLAYM1$+M1$+M2$<ENTER>
```

You should get an immediate reponse. Then:

```
FORR=1TO3:PLAYM1$+M2$:NEXT<ENTER>
```

will produce six sets of notes.

The other commands can also be stringed, and then used to form concatenations, as they are called, strings all added together, but not added in the true arithmetical sense.

For example, still in command mode:

```
T1$="T100":L1$="L32":V1$="V7"<ENTER>
PLAYT1$+L1$+V1$+M2$+"L2"+"V15"+M3$<ENTER>
```

will produce two sets of notes, one quick and quiet, one loud and slow.

Naturally these sequences can be put into a program, purely by adding line numbers:

```
10 M2$="05EDC":M1$="03CDE"
20 T1$="T100":L1$="L32":V1$="V7"
30 PLAYT1$+L1$+V1$+M2$+"R4"+"L2"+"V15"+M1$
40 END
RUN<ENTER>
```

and:

```
GOTO30<ENTER>
```

will play it again and again thereafter.

The same operators can be used, as in the **DRAW** commands, to assign variables from outside the string, for example, rewrite line 30:

```
30 PLAYT1$+L1$+V1$+M2$+"R4"+"L=D;"+"V15"+M1$
```

and add line 5:

```
5 CLS:LOCATE2,2:PRINT"Duration";:INPUTD
```

where the length of the note ' L' is made equal to 'D', the variable obtained from the keyboard.

RUN the program again, and input a few durations.

Similarly line 20 can be changed to:

```
20 T1$="T100":L1$="L=D;":V1$="V7"
```

where again 'L' is made equal to 'D'.

You can, of course, have as many 'external' variables as you have room for, for example:

```
5 CLS:LOCATE2,2:PRINT"Duration set two,
  1 to 64";:INPUTD2
6 CLS:LOCATE2,2:PRINT"Duration set one,
  1 to 64";:INPUTD1
7 CLS:LOCATE2,2:PRINT"Tempo set one,
  32 to 255";:INPUTTE
8 CLS:LOCATE2,2:PRINT"Volume set one,
  1 to 15";:INPUTVO
10 M1$="O3CDE"
20 L1$="L=D1;":T1$="T=TE":V1$="V=VO;"
30 PLAYT1$+L1$+V1$+M1$+"R4"+"L=D2"+"V15"+M1$
40 END
```

allows the values for Length, (duration), Tempo and Volume for the first set of three notes of M1$ to be chosen, and for the duration of the second set, so that comparisons can be made at the same time.

Naturally, values for S and M can also be experimented with in this way, but I shall leave you to your own devices to do this.

TWO AND THREE OF A KIND

It is also possible to play two, or even three, notes at the same time, in harmony, in other words to play two and three note chords.

Try this sequence in command mode:

```
A$="A":B$="B":C$="C":D$="D":E$="E":F$="F":
G$="G"<ENTER>
PLAYE$,G$,B$<ENTER>
```

This will produce a top quality 'car horn' sound, and you only have to type it in once to repeat it over and over again. Use the screen as a musical drawing board by moving the cursor to the PLAY line and pressing <ENTER> each time you wish to play the chord. Change the variables and experiment to get different three note chords, some I expect will be pretty dreadful!

Notice the use of the commas between the notes of the chords, but the use of the full colon when writing in the strings.

You can also write two note chords by the same method:

```
PLAYC$,E$<ENTER>
```

The notes will play in the default length and the default octave.

To change the octave type in, for example:

```
PLAY"O2"<ENTER>
```

but this will only change the first note of the chord.

All the other commands can also be used in this way but again will only change the first note of the chord, but experimenting in this way can lead to some interesting sounds.

You can also play two and three note chords by just entering the notes:

```
PLAY"E","G","B"<ENTER>
PLAY"E","B"<ENTER>
```

In fact, this sort of command mode programming can be quite exciting, try for example:

```
PLAY"O5S15M400CEG","O3S10M4000EGB",
"O7S2M12000?AAA"<ENTER>
```

which plays three three note chords with startling effect, then add L16 to the first string and play it again, and notice the difference.

You will perhaps notice now the main trouble with programming the computer to produce chords that are always in step. Rests can be used to even things up, but whatever you have done to the mini-mini-computer that looks after the sounds the computer makes it will stay that way until you change them again. This is why it is best to practice in command mode; because you can always switch off the computer and start again, and you don't lose your program, as you didn't write one.

Three part harmony is not easy to write, and requires some musical knowledge, but at least you don't have to know where all the keys are, or which holes to cover up, or even which strings to pluck! There are seven notes to learn and that's all, from then on in you are on your own, but I advise you to turn the volume down on the TV set!

To end this chapter here is a little three part harmony you might like to try, just to get you started:

```
10 COLOR10,5,1:SCREEN2,1
20 LOCATE40,80:PRINT"harmony?"
30 A1$="O4CDEFCBBO5CDEFGAO6A"
40 A2$="O4EBRGGDRCO6RCO4RARCL4C"
50 A3$="O4GGEREGDFO6GO5C+FO3AO5CAA"
60 SPRITE$(1)="cDefgAbC"
70 INTERVALON:ONINTERVAL=1GOSUB100
80 GOSUB200
90 END
100  FORX=0TO255:PUTSPRITE0,(X,89),5,1:
     FORD=1TO18:NEXT:NEXT
110 RETURN
200 PLAYA1$,A2$,A3$
210 RETURN
```

The problem for this chapter is to sort out the end of the harmony in this last program, and then you can remove the question mark in line 20!

NOTE VALUES

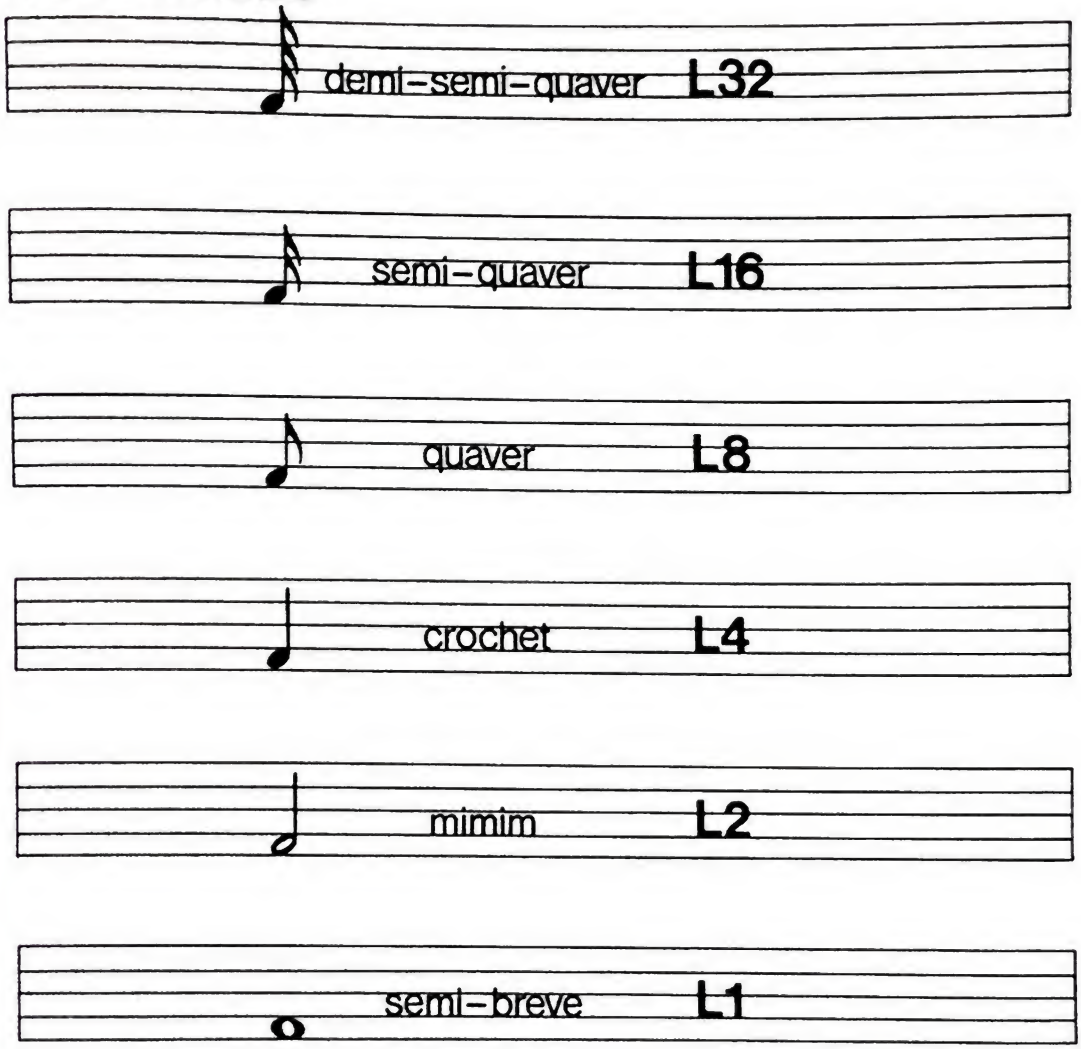


Table1

CHAPTER NINE
Synthetic Sounds

SOUND

While the command PLAY is capable of producing anything from a single note to a full blown three piece ensemble, SOUND is the special effects facility of MSX BASIC. Admittedly you can make some ghastly noises with PLAY, using the commands 'S' and 'M', and some pretty discordant blasts with the multi-channel facility, but SOUND will amaze you with its versatility.

All in all, PLAY is designed for playing music, for example:

```
PLAY"T200O4F+.E.D."<ENTER>
```

will play the opening bar of the nursery rhyme 'Three Blind Mice', and:

```
PLAY"T200O4F+.E.D.","T200D.O3A.F+."<ENTER>
```

will play it in two part harmony, while:

```
PLAY""T200F+.E.D.","T200D.O3A.F+.",
"T200O2B.B.B."<ENTER>
```

will play it in three, type this line in, and we'll experiment.

Now type in:

```
PLAY"S10M400"<ENTER>
```

and PLAY the chord again, by moving the cursor up the screen to the previous PLAY command, and pressing ‹ENTER›, and you will get a modulated or wobbly sound effect to the music, but it is still recognisable as 'Three Blind Mice'.

Change the M400 in the SM PLAY command to M4000, and repeat the PLAY chord command, and now you will get a stepping sound, but the tune is still recognisable.

Change the S10 in the SM command to S7 and you will now get a scraping sort of sound, but the tune's still there.

With the SOUND command on the other hand the sounds you get are noises, or special effects, though you may be able to get a melody out of it, it's up to you.

So with PLAY, we can say that it's music first, effects second, but with SOUND, it's effects first, and melody second.

Both the PLAY and the SOUND commands use the computer's PSG,

or Programmable Sound Generator, which in itself is a mini-mini-computer.

This computer has a number of registers, or memory areas, that can have their contents switched either on or off, or partly on and off, by setting the eights 'bits' of each register, exactly as we did when creating a sprite in an earlier chapter.

So let's see what sort of sounds we can get using the **SOUND** facility.

Once again we are going to use the command mode to carry out some experiments with the computer.

Type in the following, after first switching the computer off and on again, and then clearing the screen by pressing the CLS/HM/COPY key next to the STOP key.

```
SOUND1,Ø<ENTER>
SOUND8,Ø<ENTER>
SOUND8,15<ENTER>
```

Each command will now be separated on the screen by the word OK, and the moment you enter the last command, **SOUND,15,** you will get a clear high tone from the TV speaker. To stop the tone playing use the cursor controls to position the cursor on the **SOUND8,Ø** and press <ENTER> again, and the tone will dutifully stop!

Now using the cursor controls once more, move up to the **SOUND1,Ø** command and change this to **SOUND1,1,** and press <ENTER>, the tone will change, it will be a lower frequency now.

Carry on in this fashion, changing the **SOUND1** codes as far as the number 15, and notice the different tones you get, all reducing in frequency. **SOUND1,16** will start at 0 again and so on, so it is obvious that the codes for the **SOUND1** command are 0 to 15, giving 16 different tones.

So what have we been doing, and what do all these commands mean?

The Spectravideo, as you already know, has three sound channels, it produced a three note chord just now with the **PLAY** command.

The first channel has the SOUND commands **SOUND1** and also **SOUNDØ**, the second channel **SOUND3** and **SOUND2**, and the third channel **SOUND5** and **SOUND4**. The odd numbered channel gives a coarse frequency change when you change the code number, as I have just demonstrated, and the even numbered channel produces a fine adjustment when you change the code.

For example, type in:

```
SOUNDØ,5Ø<ENTER>
```

below the third **OK** on the screen. Now cursor up to **SOUND8,15** and press <ENTER>, and notice the fine difference to the tone played with whatever **SOUND1** has. The coarse code has a range of 0 to 255.

In this way we have 16x255 tones available, or 4080 which is quite a range. Enough, I should think, for even the most discriminating sound effects men and women.

The command **SOUND8** controls the volume for channel 1, with 0 as silence and 15 as the loudest. Try changing the value of the **SOUND8** command using the cursor controls and pressing <ENTER> while a note is playing. As you will see the control is quite fine, though finer at the top end of volume than the lower.

These then are the 4080 pure tones that MSX BASIC can produce on the Spectravideo, now let's have a look at how we can enable all the various channels, 1, 2 and 3.

To get the computer to play the sound out through the channel that we want we have to code the **SOUND7** register. Register 7 has, of course, eight bits that can be switched on or off, but bits 6 and 7 are not used, which leaves us with six to play with. Bits 3, 4 and 5 deal with the noises that the PSG can make, and bits 0, 1 and 2 deal with the tones.

To disable all the channels, the three for the tones and the three for the noise, all the bits must be switched on, that is we must use the command **SOUND7,255**.

Now type into the computer:

SOUND7,255<ENTER>

Even the key clicks have disappeared!

Try entering **SOUND8,15** commands, you won't get any sound at all, OK?

Now amend the **SOUND7** to:

SOUND7,Ø<ENTER>

you should now have your sound again. And by the way, CTRL/STOP will also switch off the sound volume registers.

As you remember, 255 in decimal is 11111111 in binary, therefore with decimal 255 each bit is set to 1, or switched on.

Bits 7 and 6 must always be on, binary 11000000, which equals a decimal number of 128+64=192, because each bit is equivalent to a decimal number, and to find the total decimal number we add all the switched on bits together. Remember:

1	1	1	1	1	1	1	1 *binary*, is:
128	64	32	16	8	4	2	1 *decimal*

Therefore:

1	1	0	0	0	0	0	0 *bits 6 and 7 on.*
128+64		0	0	0	0	0	0 *=192 decimal.*

TONE

To enable only channel 1 with a tone we must switch off bit 0, and bit 0 only. Bit 0 has the decimal value of 1, therefore to switch on channel one only we must subtract 1 from the total of 255, giving 254:

1	1	1	1	1	1	1	0 *bit 1 off only.*
128	64	32	16	8	4	2	0 *=254*

Type in now:

```
SOUND7,254<ENTER>
```

as the next line on the screen.

Channel 1 is now enabled, you won't notice any difference to the tone.

Be careful not to use both `SOUND1,Ø` and `SOUNDØ,Ø` together, or you will get no sound at all!

Experiment with these five commands now in command mode, bearing in mind the double `SOUNDX,Ø` commands.

Your screen should look like this at this stage of the experiment:

```
SOUND1,5
OK
SOUND8,Ø
OK
SOUND8,15
OK
SOUNDØ,85
OK
SOUND7,254
OK
```

But the numbers after the commas in the 1st, 3rd and 4th `SOUND` commands may well be different to mine.

NOISE

We can now start adding some noise to the tone being played by enabling the channel 1 noise facility. This is controlled by bit 3, and bit 3 has a decimal value of 8:

7	6	5	4	3	2	1	0	*bit number*
1	1	1	1	1	1	1	1	*register 7, all bits on.*
128	64	32	16	8	4	2	1	*decimal numbers in bits*
1	1	1	1	0	1	1	0	*binary arrangement for channel 1, tone and noise.*

Therefore, to switch on the noise, as well as tone, for channel 1 we must also switch off bit 3, by subtracting another 8 from the number, 254-8=246, or 9 from the full bit total of 255, 255–9=246.

We can now amend our **SOUND7** command to:

```
SOUND7,246
```

Obviously, if we only want the noise on channel 1, without the tone then we will have:

```
1   1   1   1   0   1   1   1
128 64  32  16  0   4   2   1 = 247
```

and the **SOUND7** can be changed to:

```
SOUND7,247
```

Experiment now with this **SOUND7** command, changing from 246, to 247, to 254 and 255, where:

255 - no sound at all.

254 - pure tone, channel 1.

247 - noise, channel 1.

246 - tone plus noise, channel 1.

REGISTER 6

Register 6 can make some considerable difference to the tone plus noise sound, or the noise only sound by emphasizing a particular noise frequency. For example, with an available code range of 0 to 32, we can add yet another line to our screen by:

```
SOUND6,Ø<ENTER>
```

Then changing it, with the **SOUND8** at say 15, to:

```
SOUND6,2Ø
```

and notice the difference it makes. You now have four registers to experiment with, which could keep you busy for some time!

But we are not finished yet, we can now alter the envelope of the sound being made in a similar way to the M commands we used with the **PLAY** command.

ENVELOPES

But what do I mean by the word 'envelope'?

Any sound does not, while it is being 'sounded', remain at a fixed volume. Consider a piano note for example, when it is struck by the hammer on the wire, the initial sound is very loud, a loudness that rises very quickly, called the attack period. Then it starts to decay from this maximum volume, and falls slowly away to nothing, this period is naturally called the decay period. Other instruments have different envelopes. Some have extra periods, such as a sustain period, which occurs after the decay period, where the sound remains at a reduced but fixed level for some time. Others then fall away, either quickly or slowly, from that level, after the sustain period time interval, to nothing. This is called the release period. We have, therefore, in any note some form of what is known as the 'attack, decay, sustain, release' envelope, or ADSR for short.

In MSX BASIC on the Spectravideo we have three channels that arrange for the sound to be 'enveloped', two for timing, and one for selecting the shape of the envelope.

Registers 11 and 12 look after the timing, and register 13 the shapes.

We can now, therefore, type in two more SOUND commands onto the screen:

```
SOUND11,1<ENTER>
SOUND12,1<ENTER>
```

after the SOUND6 command and OK.

These two will enable the timing registers, though SOUND12, being the coarse control, with values between 0 and 255, will not make a great deal of difference when used with a low number. SOUND11, the fine control, also has codes between 0 and 255.

The next thing to do is to enable the envelope control of the note being made. With the normal method of tone, or noise, or tone plus noise production, the sound is continuous until it is switched off with the SOUND8,Ø command, for channel 1, but with envelope control the sound made could be a one off production, depending on the envelope chosen.

To enable this fully we have to change the SOUND8 command, register 8, to SOUND8,16. The reason for this is that the envelope command controls the volume itself, depending on its shape, and a code of 16 in SOUND8 tells the computer to select the particular envelope you have selected with the code in SOUND13.

Do this now, in the usual fashion, by changing the third command on the screen.

Now complete the setting up procedure with:

```
SOUND13,1<ENTER>
```

The screen should now look like this, though some of your codes may be different to mine:

```
SOUND1,4
OK
SOUND8,Ø
OK
SOUND8,16
OK
SOUNDØ,25
OK
SOUND7,254
OK
SOUND6,Ø
OK
SOUND11,Ø
OK
SOUND12,1Ø
OK
SOUND13,12
OK
```

Newcomers to computing, especially using the command mode of addressing the computer, will find that things can very quickly go wrong.

If the screen does get out of hand, or if you want to impress your friends with MSX BASIC sound effects, then switch off first, and type in this experimental screen, and away you go!

What you have now is a 'command sound generator', controlled only by the cursor controls, the keyboard, and the <ENTER> key. You can produce any sort of sound, with or without noise, using 4080 different tones, and nine different envelopes, with a vast array of timing sequences, either one off or continuous.

When you have the sound you are looking for, all you have to do is write down the various SOUND commands you need, and put them into your program.

Always make sure that the **SOUND8,Ø** command comes at the end of the sound you want to make, otherwise the sound will continue playing, unless, of course, it has been enveloped to only produce one blast of sound, such as **SOUND13** commands 0, 1, 2, 3, 4, 5, 6, 7, 9 or 15.

ENVELOPE SHAPES

I will now describe the various sounds you can get by using the envelope shapers 0 to 15.

0 - instant attack, slow decay, one pulse only.

1 - instant attack, slow decay, one pulse only.

2 - instant attack, slow decay, one pulse only.

3 - instant attack, slow decay, one pulse only.

4 - medium attack, instant decay, one pulse only.

5 - medium attack, instant decay, one pulse only.

6 - medium attack, instant decay, one pulse only.

7 - medium attack, instant decay, one pulse only.

8 - instant attack, medium decay, repeated continuously.

9 - instant attack, slow decay, one pulse only.

10 - instant attack, gradual decay, small sustain, slow release, repeated continuously.

11 - instant attack, very quick decay, instant second rise to a sustained full volume.

12 - medium attack, instant decay, repeated continuously.

13 - slow attack, sustained full volume.

14 - slow attack, no decay, small sustain, slow release, repeated continuously, sine wave pattern.

You may think that you will need eight **SOUND** commands to produce a sound, which may seem a lot, but you can use less. For example **SOUND0** is not really needed, unless you need to fine tune. **SOUND6** is not needed unless you need to make a particular noise frequency predominant over all the others. **SOUND11** is not needed either, unless again you need to fine tune the selected repetition time of the continuous envelopes.

And when using the envelope patterns, if a **SOUND1** command is not used, the computer will default to a given low frequency value, and you will still get a sound produced.

We could therefore reduce our experimental screen to:

```
SOUND8,0<ENTER>
SOUND1,5<ENTER>
SOUND8,16<ENTER>
SOUND7,254<ENTER>
SOUND12,100<ENTER>
SOUND13,14<ENTER>
```

to produce a whole range of sounds, either enveloped or not.

The two **SOUND** commands, 11 and 12, control the number of times that a pattern will be repeated per second, with 255 the slowest and 0 the quickest meaning that theoretically we have 256x255 variations of speed. A total of 65280, that's enough for anyone!

The above screen layout will produce a vibrating gong-like, rising and falling pattern, reminiscent of the Doppler effect, well loved of physics teachers.

Try changing the values in **SOUND12**, and you will see what I mean about the repetition of the pattern.

Change SOUND1 to SOUND1,0 and you will have a tiny bell with feet running past your window, Wee Willie Winkie?

Change it to SOUND1,15 and you will have a Jewish harp instrument.

Have fun, that's what home computing is all about!.

CHANNELS 2 AND 3

We now know how to program channel 1 to produce a variety of sounds, but there are two other channels we can use, channels 2 and 3.

Channel 2 has the tone select register of SOUND3 and SOUND2, 3 for the coarse tuning, and 2 for the fine.

Channel 3 has the tone select register of SOUND5 and SOUND4, 5 for the coarse tuning, and 4 for the fine.

Channel 2 has SOUND9 for its amplitude, and channel 3 has SOUND10.

The noise predominant frequency of SOUND6 applies to all channels, as does the envelope repetition frequency select of SOUND11 and SOUND12, and the envelope shape select of SOUND13.

REGISTER 7

This leaves SOUND7, the enable channel register.

To enable the correct channel we must calculate what code number, X, to place after the comma in SOUND7,X.

As I mentioned before we are only allowed to use the first 6 bits of this register, and must therefore always leave bits 6 and 7 switched on. Not much will happen to the sound if we do switch them off, but something may be happening elsewhere if we do, like the IN and OUT ports of the computer, and we have enough gremlins and bugs in computing already, so leave them on!

To do this, the code number must always be at least decimal 192, 128+64. This leaves a possible 64 variations to enable all the other tone and noise channels, either together or on their own.

To find the correct number for the channels of tone or noise that you may want for SOUND7, use the following table:

bit	5	4	3	2	1	0
channel	N3	N2	N1	T3	T2	T1
code number	32	16	8	4	2	1

Where N stands for Noise, and T for Tone.

Decide which channel or channels you wish to enable, and whether you

want tone, noise, or both. Add up the numbers for each bit needed to be enabled, and subtract the total from 255.

For example:
255–0=255 - all channels off.

255–1=254 - channel 1, tone only.

255–2=253 - channel 2, tone only

255–3=252 - channels 1 and 2, tone only.

255–6=249 - channels 3 and 2, tone only.

255–7=248 - channels 1, 2 and 3, tone only.

255–8=247 - channel 1, noise only.

255–9=246 - channel 1, tone and noise.

255–18=237 - channel 2, tone and noise.

255–63=192 - channels 1, 2 and 3, tone and noise.

MULTI-CHANNEL SOUND

Now clear the screen, which is equivalent to using NEW when you have a program, and type in the following commands in command mode:

```
SOUND1,2:SOUND3,7:SOUND5,15<ENTER>
```

This will place a high frequency tone in channel 1, a medium frequency note in channel 2, and a low frequency note in channel 3.

```
SOUND8,16:SOUND9,16:SOUND10,16<ENTER>
```

This will switch the sound registers of the three channels to 'envelope control'.

```
SOUND7,248<ENTER>
```

Will enable the three channels for tone only.

```
SOUND12,10<ENTER>
```

Eventually produces a continuous medium slow repeated sound.

```
SOUND13,14<ENTER>
```

Produces a continuous sound of slow attack, small decay, no sustain and slow release, in a sine wave pattern. The sound you should be getting is somewhat similar to a faulty motorbike, with a loose metal mudguard.

```
SOUND8,0<ENTER>
```

Allows you to switch off channel 1 only, which fixes the metal mudguard.

```
SOUND9,0<ENTER>
```

Allows you to switch off channel 2, which makes you wish the darned thing would rev up and disappear.

```
SOUND10,0<ENTER>
```

Which is what it now does, channel 3 is switched off too.

```
SOUND8,0:SOUND9,0:SOUND10,0<ENTER>
```

This line allows you to switch off all three channels at once, the motorbike has suddenly run out of petrol!

Your screen should now look like this, which will allow you to experiment with one, two, or three channels either mixed or independently, using either tone, noise or both:

```
SOUND1,2:SOUND3,7:SOUND5,15
OK
SOUND8,16:SOUND9,16:SOUND10,16
OK
SOUND7,248
OK
SOUND12,10
OK
SOUND13,14
OK
SOUND8,0
OK
SOUND9,0
OK
SOUND10,0
OK
SOUND8,0:SOUND9,0:SOUND10,0
OK
```

Here are a few ideas for you, but in the main you are on your own. Don't forget that if you get the screen in a mess, use the cursor controls to line it all up again. If, by chance, you <ENTER> a wrong code number,

500 instead of 50 with `SOUND13` for example, you'll get an error code, the left arrow key will wipe out the words you don't want, as will the delete key, or just use the space bar as a screen rubber.

Try these to get you going, the possibilities are endless, especially if you type in the extra `SOUND6` register, which has a range of 0 to 31, and force a particular noise frequency through.

Change `SOUND7` to `SOUND7,192` to get all three noises as well, which will sound a bit like Dad filing the dining room table legs.

Change `SOUND12` to `SOUND12,2,` does this sound like an old fashioned steam train?

Switch out the channels one by one, and the train moves off down the line.

Change SOUND12 to 200, and SOUND7 to 232, noise only on channel 2, and you get the chap next door with his motor mower.

Change SOUND7 to SOUND7,239, SOUND12 to SOUND12,255, and SOUND13 to SOUND13,10, 'puffing billy' has at last arrived at the station, but it's raring to go again.

Change SOUND12 to SOUND12,100, and it has!

Change SOUND9 to SOUND9,15, and someone keeps turning the waterfall on and off, switch off channel 1 and 3, and it's on all the time.

As I said, the experiments are limitless.

All these sounds of course can be put into programs as required.

FOR....NEXT loops are very useful to repeat single sounds or continuous ones that are broken up by a SOUND off call, SOUND8,0 for example.

For example type in this short demonstration program:

```
10 COLOR4,9,1:SCREEN2,0
20 LOCATE40,80:PRINT"mixtures"
30 SOUND7,254:PLAY"T20004FGAB-05CCDEFEDC04B-
   AGF":FORD=1TO3700:NEXT
```

Line 30 PLAYs the F Major scale, up and down, but has to have quite a long delay loop as the music computer plays independently of the rest of the computer's operating system. SOUND7,254 is there to enable the correct sound channel again, because this demonstration program, as you will see, is in a continuous loop.

```
40 SOUND7,192:SOUND0,225:SOUND1,12:SOUND8,15
50 FORD=1TO 2000:NEXT:SOUND8,0
```

These two lines play a motorbike or motor boat sound for 2000 units worth, before switching off the sound channel.

```
60 GOSUB200
```

A short gap between demonstrations.

```
70 SOUND1,15:SOUND8,16:SOUND3,10:SOUND9,16
80 SOUND12,30:SOUND13,12
90 FORD=1TO2500:NEXT:SOUND9,0
```

These three lines produce the whirr of helicopter blades for a while, using two channel enveloped sound. After which channel 2 is switched off.

```
100 SOUND12,100:SOUND7,254:SOUND1,1:
    SOUND13,11
```

```
110 FORD=1TO1250:NEXT
```

The dinner gong!

```
120 SOUND7,247:SOUND6,8:SOUND13,8:SOUND12,30
130 FORD=1TO3600:NEXT:SOUND7,255:SOUND8,0
140 GOSUB200:GOTO30
```

Six gun shots blast out, and round we go again, after switching everything necessary off.

```
150 END
200 FORD=1TO1000:NEXT:RETURN
```

Line 150 stops the program from running into the subroutine, and the subroutine for the short delay is in line 200.

Thank goodness for **FOR....NEXT** loops!

That is the end of this brief two chapter look at the sound facilities in MSX BASIC, and on the Spectravideo. Hopefully what I have shown you has just whetted your appetite. Now for some special effects of a third kind and they are waiting for you in the next chapter.

CHAPTER TEN
Screen effects

In this chapter I shall be discussings ways of using the computer to produce various screen effects, which when associated with the special effects in the last chapter, could produce some interesting highlights into your programs. The demonstrations I am going to show you can be endlessly adapted to create designs of your own invention.

When you are developing your programs, it is always a good idea to be able to flash back and forth between the listing and the screen display, without having to stop the program and type `LIST`.

To achieve this use the following routine, which uses the command `ONKEY`, but as this command, on the Spectravideo, can't be used with `GOTO`, we have to cheat and use `GOSUB`.

```
1000 KEYON:ONKEYGOSUB1020
1010 GOTO1010
1020 CLS:SCREEN0,0:LIST
```

CATERPILLAR

```
10 COLOR1,4,7:SCREEN1
```

Sets up the screen.

```
20 X=40:Z=X:FORBUMP=1TO8
30 LINE(X-8),95)-(X+15,96),1
40 CIRCLE(X,96),40,1,0,22/7,5/1
50 PAINT(X,94),1:X=X+17:NEXT
```

These four lines draw eight, black filled, half circles across the screen, the caterpillar's 'BUMPS'.

```
60 CIRCLE(Z-10,106),106,10,12
70 CIRCLE(X+25,104),12,10
```

Draws the outline of the caterpillar's rear end and head in different colours.

```
80 PAINT(28,98)12:PAINT(203,98),10
```

Fills them in, in green and yellow.

```
90 LINE(30,96)-(200,116),12,BF
```

Draws the caterpillars's body with a filled in green rectangle.

```
100 LOCATE206,94:COLOR4:PRINT".":COLOR1
```

Gives the caterpillar a blue eye.

```
1000 KEYON:ONKEYGOSUB1020
1010 GOTO1010
1020 CLS:SCREEN0,0:LIST
```

This is the program development routine mentioned above.

This program shows that you can have two different colours adjacent to each other, but you have to be careful where you start to **PAINT** in each case. Line 50 ensures that the **PAINT** start point is well inside each circle.

It also shows how you can develop a screen display by painting one colour on top of another to simplify the mechanics of the drawing program. For example, if I had drawn the caterpillar's body before the head, because the head is in a different colour, the head would have been filled in on top of the body. The way I have done it simplifies the circle command required, in that I have drawn a full circle and then covered the part I don't want with the body. Swop lines 90 and 60, which will demonstrate what I mean, and also show you how the whole screen can produce quite unexpected effects if you get the order or location codes wrong. This program, after the line swop, (and don't forget to use the screen editor to do this), draws the bumps, tail and head, then fills the screen with green, and then yellow, finally ending up with a yellow screen, a green circle and a blue dot!

SQUARE WAVES

This short demonstration program will give you the opportunity to experiment with making rectangles and filling them in, and to see how the nearness, or otherwise, of one block of colour to another affects the ways in which the blocks are filled. It also gives you a program to draw unlimited square waves, depending on the width, height and number of waves required.

When the program **RUN**s, input the width of your square wave, and the height, bearing in mind that the number inputted will be used as a reciprocal. Then input the start point, which will show how the first block is affected by its nearness to another colour, and then input the number of waves required. Next you are asked for the colour for the top of the square wave, and then for the bottom of the square wave.

```
10 COLOR1,4,6:SCREEN0,0
20 LOCATE4,4:PRINT"WIDTH, X, 5 TO 50";:INPUTX
30 CLS:LOCATE4,4:PRINT"HEIGHT, 1/Y,
   Y=20 TO 90";:INPUTY
40 CLS:LOCATE4,4:PRINT"START POINT, Z,
   50 TO 80";:INPUTZ
50 LOCATE4,4:PRINT"NUMBER OF WAVES, W,
   1 TO 20";:INPUTW
60 CLS:LOCATE4,4:PRINT"COLOUR WAVE TOP, C1,
   0 TO 15";:INPUTC1
70 LOCATE4,4:PRINT"COLOUR WAVE BOTTOM, C2,
   0 TO 15";:INPUTC2
80 SCREEN1
90 LINE(50,96)-(240,96):LINE(50,5)-(50,185)
```

```
100 FORR=1TOW
110 LINE(Z,Y)-(Z+X,96),C1,BF
120 LINE(Z+X,96)-(Z+2*X,96+96-Y),C2,BF
130 Z=Z+2*X:NEXT
1000 KEYON:ONKEYGOSUB1020
1010 GOTO1010
1020 CLS:SCREEN0,0:LIST
```

The 'F' can be removed from the 'BF' in either line 110 or 120, or both, to show what happens to the colours when only lines are drawn.

A similar sort of program to this one could be used to draw bar charts or histograms, by the suitable adjustment of the X and Y parameters.

SLOW GROWTH

In this demonstration a circle grows upwards to form a cylinder, giving a good example of how screen figures can be made to slowly increase in size.

```
10 COLOR1,4,6:SCREEN1,0
20 FORY=120TO40STEP-1
30 CIRCLE(88,Y),20,6:PAINT(88,Y),6
40 FORD=1TO30:NEXT:NEXT
50 CIRCLE(88,Y),20,1
1000 KEYON:ONKEYGOSUB1020
1010 GOTO1010
1020 CLS:SCREEN0,0:LIST
```

It will be noted that when the circle has finished growing, and the end or top of the cylinder is drawn with an unfilled circle, other areas of the shape are filled in as well. This again is due to the computer filling in adjacent areas of screen display and this is because some pixels affect other nearby pixels. The position of the shape on the x axis, and the Y axis, can alter and sometimes improve this difficulty; it will be up to you to experiment to get the best display for your screen.

SPUTNIK

This growth routine can be used to design a 'sputnik' or satellite:

```
10 COLOR1,4,6:SCREEN1,0
20 DRAW"BM128,96C15NU40NR40ND40NL40"
30 FORRADIUS=0TO20
40 CIRCLE(128,96),RADIUS,6:PAINT(128,96),6:
   NEXT
1000 KEYON:ONKEYGOSUB1020
1010 GOTO1010
1020 CLS:SCREEN0,0:LIST
```

With the addition of a few extra lines it could also be the North Star.

```
25 DRAW"NE30NF30NG30MH30"
```

providing the aspect is changed by amending line 40 to suit:

```
40 CIRCLE(128,96),RADIUS,6,,,1.4/1:
   PAINT(128,96),6:NEXT
```

By swapping round the order of the lines of the program, you can demonstrate once again the 'pixel creeping' effect of the screen:

```
10 COLOR1,4,6:SCREEN1,0
20 FORRADIUS=0TO20
30 CIRCLE(128,96),RADIUS,6:PAINT(128,96),6:
   NEXT
40 DRAW"BM128,96C15NU40NR40ND40NL40"
50 DRAW"NE30NF30NG30NH30"
```

The order in which screen graphics commands are carried out in a program is very important to the final visual look of the screen.

By changing slightly the locations of the **DRAW** command, (126,98), you will see that in fact it is the vertical and diagonal lines that have the worst effect on the display. Horizontal lines do not incur 'pixel creep'.

To demonstrate this delete line 50 from the above program, and **RUN** it again, only the vertical line now has an effect.

Next delete the '**U40**' movement command from the DRAW command in line 40, and **RUN** the program, note the difference.

Lastly, delete the '**D40**' command as well, RUN and take note.

No pixel creep.

Swap the order of the lines again, the **DRAW** before the **CIRCLE**, and you will notice how in this location of the DRAW command parts of the white line are unchanged as the circle grows. Change the location of the **DRAW** command back to (128,96), and all the line within the circle will be removed or recoloured.

Now copy the **DRAW** command into the loop, and **RUN** the program. You will notice the **CIRCLE** changing the pixel colours in the line as it is drawn.

Now replace the '**U40**' and '**D40**' commands back into the two **DRAW** commands, and **RUN** the new program. You will notice now the pixel creep in the vertical axis.

Finally, change the first **DRAW** command to (128,96), the original location, and the centre of the circle. Change the **DRAW** command in the loop to (120,92), and the colour code to black, 1. Then increase the area of the circle by increasing the loop variable to 30, and re**RUN** the program. This should complete the picture.

If you have the time put back the diagonals!

SQUARE GROWTH

The following short program will draw a thick black line on the screen:

```
10 COLOR1,4,6:SCREEN1,0
20 X=100
30 FORR=1TO40:LINE(X,80)-(X+1,84),10,B
40 X=X+1:NEXT
1000 KEYON:ONKEYGOSUB1020
1010 GOTO1010
1020 CLS:SCREEN0,0:LIST
```

Changing lines 20 and 30 to:

```
20 X=100:Y=80
30 FORR=1TO40:LINE(X,Y)-(X+1,Y+1),10,B
40 X=X+1:Y=Y+1:NEXT
```

will draw a thick diagonal line, SCREEN2 ON SCREEN1.

The increments will determine the thickness of the line in the first program, and the steps and angle of the diagonal, eventually producing dotted diagonals, in the amended one.

Changing the sign of the increments wil determine in which direction the line is drawn, and including a time delay loop will affect the speed the line is drawn. Changing the value of the number added to the X and Y in the second bracket in the **LINE** command in line 30 will increase the thickness of the line.

This now naturally leads to the drawing of cubes. Change the following lines:

```
20 X=100:Y=80
30 FORR=1TO25:LINE(X,Y)-(X+20,Y+30),1,B
40 X=X+1:Y=Y+1:NEXT
```

Line 20 determines where on the screen the cube will appear, or where the thick lines will start from.

Line 30 draws the cube or the lines.

Line 40 looks after the angles at which the cube or lines appear by changing the sign of the increments, and can have an effect on the picture by staggering the repeat.

Change line 40 to:

```
40 X=X+4:Y=Y-2:NEXT
```

Add 'F' to the LINE command and the staircase will be filled in.

Change the screen mode to SCREEN2, and you'll get a concertina!

A CALL FOR YOU

The ring of the telephone bell can be simulated by using the third sound facility of the computer, the BEEP command.

```
10 COLOR1,4,6:SCREEN2,0
20 FORTELE=1TO4
30 FORRING=1TO2
40 FORBELL=1TO10:BEEP:NEXT
50 FORQUIET=1TO200:NEXT
60 NEXT
70 FORQUIET=1TO500:NEXT
80 NEXT
90 LOCATE60,80:PRINT"HELLO?"
1000 KEYON:ONKEYGOSUB1020
1010 GOTO1010
1020 CLS:SCREEN0,0:LIST
```

Here, by the use of five nested `FOR....NEXT` loops, I have recreated the necessary sound, quite realistically, I think.

The whole bell ringing routine could, of course, be written on one program line, but I have split it up to indicate it's construction.

```
10 COLOR1,4,6:SCREEN2,0
20 FORTELE=1TO4:FORRING=1TO2:FORBELL=1TO10:
   BEEP:NEXT:FORQUIET=1TO200:NEXT:NEXT:
   FORQUIET=1TO500:NEXT:NEXT
90 LOCATE60,80:PRINT"HELLO?"
```

STAIRS

Stairs or steps can be drawn without complicated DRAW or LINE statements, providing the routine is RUN at the beginning of the program, by partly clearing the screen to some other background colour, or one to match the border.

```
10 COLOR2,1,6:SCREEN1,0
20 Y=8:X=30
30 FORR=1TO39:LINE(X,Y-8)-(256,Y),6,BF
40 Y=Y+8:X=X+5:NEXT
50 SPRITE$(1)="ZZZZZZZZ":X=143
60 FORY=1TO178TO0STEP-8
70 PUTSPRITE0,(X,Y),15,1
80 FORD=1TO200:NEXT
90 X=X-5:NEXT
100 FORD=1TO300:NEXT
110 FORR=XTOX+210
120 PUTSPRITE0,(R,Y+6),15,1
130 FORD=1TO30:NEXT:NEXT
1000 KEYON:ONKEYGOSUB1020
1010 GOTO1010
1020 CLS:SCREEN0,0:LIST
```

This program will draw a set of stairs from half way across the bottom of the screen to the top of the viewing screen. At the top is a long tube. A white ghost, (sprite), climbs the stairs, and runs along the tube.

Line 30 draws the stairs and the tube, the rest looks after the sprite.

CURTAINS

The previous routine can be adapted to lower the curtain on the screen, which I think is a fitting end to the chapter:

```
10 COLOR1,6,1:SCREEN1,0
20 X=0:Y=0
30 FORR=0TO175:LINE(X,Y)-(256,Y+1),1,BF
40 Y=Y+1
50 FORD=1TO10:NEXT
60 NEXT
70 LOCATE110,181:PRINT"THE END"
80 FORD=1TO 1000:NEXT
90 X=0:Y=176
100 FORR=0TO14:LINE(X,Y)-(256,Y+1),1,BF
110 Y=Y+1
120 FORD=1TO20:NEXT
130 NEXT
1000 KEYON:ONKEYGOSUB1020
1010 GOTO1010
1020 CLS:SCREEN0,0:LIST
```

CHAPTER ELEVEN
A change of face

Computer graphics and sound on modern home computers have extensive facilities, and MSX BASIC on the Spectravideo is no exception.

Graphics can be drawn extremely easily, with little mathematical knowledge, by complete novices with a good knowledge of BASIC.

Music requires some musical expertise, but is not difficult to master, once the fundamental musical theory is understood. Speech, though, is still in its infancy, and the spoken word using MSX BASIC, at the moment of writing, is not readily available.

But you can take a short cut, and by using the cassette control commands of the language, input speech into your programs.

The two commands to do this are **MOTOR ON** and **SOUND ON**, and their complements **MOTOR OFF** and **SOUND OFF**, which control the motor and audio channel of the cassette recorder respectively.

In the following program I have made arrangements for the graphics to follow a well known modern song 'Forty and Fadin', by J.J. Barrie, available on Magic Records from your local record shop.

If this program is loaded into the computer from tape, after you have typed it all in and saved it, and then the audio tape is placed in the cassette recorder, and started when the program is **RUN**, the graphics should coincide with the story line of the song.

An alternative is to record the song at the end of your computer program on the same tape, and then load the program. When the program is RUN the cassette recorder will automatically start and you will hear the song through the TV loudspeaker.

I have only used those commands and statements discussed in the chapters of this book. In a nutshell, the program draws a face, which slowly, in time with the words of the song, grows older, until at the end the old man cries for his lost youth.

To help in understanding the program I have broken it into parts, each with a short explanation.

```
10 MOTORON:SOUNDON
20 COLOR1,14,14:SCREEN1,0
30 REM OUTLINE OF FACE
40 CIRCLE(128,90),80,9,,,1.5/1:
   PAINT(128,96),9
50 REM EYELIDS
60 CIRCLE(152,80),10,15,,,1/3:
   CIRCLE(104,80),10,15,,,1/3
70 PAINT(152,80),15:PAINT(104,80),15
```

```
80 REM EYES
90 CIRCLE(152,80),10,1,,,1/3:
   CIRCLE(104,80),10,1,,,1/3
100 REM TIME DELAY:GOSUB1730
110 REM MOUTH
120 CIRCLE(128,137),22,15,6.28,3.14,1/6
130 CIRCLE(128,137),22,15,3.14,6.28,1/5
140 PAINT(128,137),15:GOSUB1730
150 REM NOSE
160 LINE(130,120)-(140,120),1,B:LINE(116,120)
    -(126,120),1,B:LINE(128,70)-(128,120),1
170 CIRCLE(128,120)12,1,2.62,3.14:
    CIRCLE(128,120),12,1,6.28,0.52
180 LINE(138,118)-(130,70),6:
    LINE(118,118)-(126,70),6:GOSUB1730
190 REM EYEBROWS
200 CIRCLE(152,75),12,1,6.28,3.14,1/4
210 CIRCLE(152,75),14,1,6.28,3.14,1/4
220 CIRCLE(152,75),16,1,6.28,3.14,1/4
230 CIRCLE(152,75),12,1,6.28,3.14,1/4
240 CIRCLE(152,75),14,1,6.28,3.14,1/4
250 CIRCLE(152,75),16,1,6.28,3.14,1/4:
    GOSUB1730
260 REM PUPILS
270 CIRCLE(152,80),2,13:CIRCLE(104,80),2,13
280 PAINT(152,80),13:PAINT(104),13
290 CIRCLE(152,80),1,1:CIRCLE(104,80),1,1
300 REM EARS
310 LINE(176,71)-(195,155),14,BF:LINE(64,71)
    -(78,155),14,BF:GOSUB1730
320 LINE(176,68)-(176,130),6:LINE(78,68)-
    (78,130),6:GOSUB1730
330 CIRCLE(78,100),30,6,1.37,4.81,4/1:
    CIRCLE(176,100),30,6,4.61,1.45,4/1
340 PAINT(77,100),6:PAINT(179,100),6:
    GOSUB1730
350 REM HAIR
360 GOSUB1760
370 FORR=1TO3:GOSUB1730:NEXT
```

This part of the program draws the complete 'young' face of the man. If you wish to **RUN** it, it is necessary to type in the two subroutines, one at 1730, the time delay, and the other at 1760, the routine to draw the hair.

The repeated reference to the delay subroutine at 1730 is to keep the program in time with the song, if you wish to dispense with the music and the song then you can leave out all the GOSUB 1730's.

```
380 REM FIRST LINES UNDER EYES
390 CIRCLE(152,80),18,6,3.92,6,1/3
400 CIRCLE(104,80),18,6,3.14,5.5,1/3
410 GOSUB1730:GOSUB1730
```

The first part of the ageing process!

```
420 REM GLASSES
430 LINE(72,68)-(178,70),12,BF
440 LINE(125,68)-(123,85),12,BF
450 LINE(131,68)-(133,85),12,BF
460 LINE(171,68)-(169,85),12,BF
470 LINE(85,68)-(87,85),12,BF
480 CIRCLE(105,85),20,12,3.14,6.28,1/4
490 CIRCLE(105,86),20,12,3.14,6.28,1/4
500 CIRCLE(151,85),20,12,3.14,6.28,1/4
510 CIRCLE(105,86),20,12,3.14,6.28,1/4
```

And the next stage, his eyesight is beginning to fail.

```
520 REM BLACK MOUSTACHE
530 CIRCLE(128,137),27,1,6.28,3.14,1/4
540 CIRCLE(128,137),27,1,6.28,3.14,1/2
550 PAINT(128,129),1:GOSUB1730
560 REM BLACK BEARD
570 CIRCLE(128,96),55,1,3.14,6.28,1.2/1
580 CIRCLE(128,98),93,1,3.76,5.66,4.2/1
590 CIRCLE(128,98),90,1,3.76,4.67,2.2/1
600 CIRCLE(128,98),90,1,4.75,5.66,2.2/1
610 CIRCLE(128,98),80,1,3.7,4.3,1.5/1
620 CIRCLE(128,98),80,1,5.1,5.72,1.5/1
630 LINE(80,90)-(82,140),1,BF
640 LINE(174,90)-(176,140),1,BF
650 LINE(105,143)-(151,158),1,BF
660 PAINT(128,160),1:PAINT(104,160),1:
    PAINT(157,158),1:PAINT(157,143),1
670 FORR=1TO3:GOSUB1730:NEXT
```

Life suddenly takes on a different aspect, as a beard is grown, and a position of authority is achieved.

```
680 REM THE FIRST BALD PATCH
690 CIRCLE(128,11),25,9,,,1/3:PAINT(128,10),9
700 REM GREY HAIR
710 GOSUB1190
```

Now the corner has been turned, and the man is on the downward slope, his hair is turning grey, and next the lines start to form on his face.

```
720 REM FROWN LINES
730 CIRCLE(128,52),36,6,6.28,3.14,1/8
740 CIRCLE(128,57),36,6,6.28,3.14,1/8
750 CIRCLE(128,62),36,6,6.28,3.14,1/8
760 CIRCLE(104,96),16,6,3.14,6.28,1/2
770 CIRCLE(152,96),16,6,3.14,6.28,1/2
780 CIRCLE(104,96),32,6,4.19,5.23,1/2
790 CIRCLE(152,96),16,6,3.14,6.28,1/2
800 GOSUB1730:GOSUB1730:GOSUB1730
```

Now his hair begins its greying process, and the end is at last in sight, forty is around the corner.

```
810 REM GREYING HAIR
820 CIRCLE(128,86),85,14,6.28,3.14,1.4/1
```

```
830 CIRCLE(78,100),32,14,1.37,3.14,4/1
840 CIRCLE(128,90),85,9,0.9,2.24,1.5/1:
    GOSUB1730
850 LINE(128,146)-(128,190),14
860 CIRCLE(125,168),20,14,1.57,4.71,8/1
870 CIRCLE(131,168),20,14,4.71,1.57,8/1
880 CIRCLE(120,164),18,14,1.57,4.71,8/1
890 CIRCLE(136,164),18,14,4.71,1.57,8/1
900 CIRCLE(115,164),18,14,1.57,4.71,8/1
910 CIRCLE(141,164),18,14,4.71,1.57,8/1
920 CIRCLE(110,160),14,14,1.57,4.71,8/1
930 CIRCLE(146,160),14,14,4.71,1.57,8/1
940 CIRCLE(105,155),11,14,1.57,4.71,8/1
950 CIRCLE(151,155),11,14,4.71,1.57,8/1
960 CIRCLE(100,152),9,14,1.57,4.71,8/1
970 CIRCLE(156,152),9,14,4.71,1.57,8/1
980 CIRCLE(128,137),27,14,0.05,3.09,1/3
990 CIRCLE(128,137),27,14,0.05,3.09,1/2.3
1000 CIRCLE(95,147),9,14,1.57,4.71,8/1
1010 CIRCLE(161,147),9,14,4.71,1.57,8/1
1020 CIRCLE(90,142),8,14,1.57,4.71,8/1
1030 CIRCLE(166,142),7,14,4.71,1.57,8/1
1040 CIRCLE(171,132),10,14,4.71,1.57,8/1
1050 CIRCLE(85,132),8,14,1.57,4.71,8/1:
     GOSUB1730
1060 REM MOUTH CHANGE
1070 CIRCLE(128,137),22,6,6.28,3.14,1/6
1080 CIRCLE(128,137),22,6,3.14,6.28,1/5:
     PAINT(128,137),6
1090 FORR=1TO3:GOSUB1730:NEXT
```

Age affects all parts of our body.

```
1100 REM WHITE TEARS
1110 S$=CHR$(0)+" pppp ":SPRITE$(1)=S$
1120 FORR=80TO196
1130 PUTSPRITE0,(104,R),15,1:PUTSPRITE1,
     (152,R),15,1
1140 PUTSPRITE2,(104,R+30),15,1:PUTSPRITE3,
     (152,R+30),15,1
1150 FORX=1TO15:NEXT
1160 IFR=160GOTO1120
1170 Q=Q+1:IFQ=500GOTO1800
1180 NEXT
```

Even men are allowed to cry, but he still looks pretty good!

```
1185 REM
1190 REM GREYING HAIR:GOSUB1730:W=10
1200 FORY=1TO66STEP2
1210 FORX=WTO200STEP2
1220 PSET(X,Y),14:NEXTX:W=W+1:NEXTY:RETURN
```

Something that happens to us all, in time.

APPENDIX ONE
BASIC Mathematics

As with any computer langauge there are many and various mathematical functions in the MSX BASIC language used on the Spectravideo computer.

In this appendix the majority of them are explained, in alphabetical order, together with a few examples of their use.

ABS(n)

Returns the ABSolute value of the expression n, that is, turns negative numbers into equivalent positive numbers, but leaves positive numbers unaffected. For example, the absolute value of -79.8 is 79.8, and the absolute value of 79.8 is again 79.8. It can be used to calculate the difference between two numbers where it is not known which of the two numbers is the greater. For example, expressing the numbers as variables, A-B will always be positive if A is greater than B, but will be negative if and when B is greater than A. But, using the **ABS** function, the computer will always return a positive value to the function ABS(A-B), where (A-B) is equal to (n), regardless of which is the greater of the two numbers.

ATN(n)

This function calculates the value in radians of an angle expressed as a tangent.

CINT(n), INT(n) and FIX(n)

These three functions convert an expression n into an INTeger, that is remove the decimal part of the expression, and therefore return a whole number less than the number supplied or calculated. For example:

```
PRINT CINT(50/1.008)<ENTER>  returns 49, as does:
PRINT INT(50/1.008)<ENTER>   and:
PRINT FIX(50/1.008)<ENTER>   whereas:
PRINT 50/1.008<ENTER>  returns 49.603174603174
```

FIX(n), if the expression n has a value greater than 15 digits, may return a whole number 1 greater than the expression, in other words rounds up the expression.

COS(n)

This function calculates the COSine of an angle (n) that has been expressed in radians.

CSNG(n) and CDBL(n)

These two functions convert numbers between single and double precision respectively, for example:

```
n=50/1.008<ENTER>
PRINT CSNG(n)<ENTER>  returns 49.6032
PRINT CDBL(n)<ENTER>  returns 49.603174603174
PRINT CSNG(49.6031746031746)<ENTER>  returns 49.6032,
```

but of course:

`PRINT CDBL(49.6Ø32)<ENTER>` returns 49.6032

Single precision numbers contain only 7 digits including the decimal point, and double precision numbers contains 15 digits. Use of the CSNG function rounds up to the nearest number for the last number of the expression. A single precision number uses 4 bytes of memory, and a double precision uses 8 bytes, whereas an integer uses only 2.

Remember that n, a number depending on the use of `CSNG` or `CDBL`, and n%, an integer or whole number, are two different variables as far as the computer is concerned. In the above example `PRINT n%` will produce 0, whereas `PRINT n` will produce 49.603174603174, and `PRINT INT(n)` and `PRINT CINT(n)` will both return 49. If you wish to use a number expressed only as an integer, then the variable must be followed by the symbol %, and conversions on it with `CSNG` and `CDBL` will always be returned as a whole number. That is:

```
n%=5Ø/1.ØØ8<ENTER>
PRINT n%<ENTER> returns 49
PRINT CDBL(n%)<ENTER> returns 49
PRINT CSNG(n%)<ENTER> returns 49.
```

EXP(n)

This function calculates e, where e=2.7183...., raised to any power n, but the calculation is performed on a double precision value for e.

LOG(n)

This function returns the natural or common logarithm of the expression n to base 10.

(n1)MOD(n2) and (n1)\(n2)

`MOD` returns the remainder after an integer division, that is when using whole numbers. If non integer numbers are used the computer will truncate the number to produce an integer before performing the operation on them.

\, more commonly known as `DIV`, returns the whole number part of an integer division. For example:

`PRINT 2ØMOD3<ENTER>` returns 2, the remainder, whereas:
`PRINT 2Ø\3<ENTER>` returns 6, as:
20 divided by 3 equals 6, with 2 remaining.

Do not confuse the \ symbol with the / symbol, the latter performs normal untruncated arithmetic, that is:

PRINT 20/3=6.6666666666667

SGN(n)

Returns +1, 0 or -1 depending on the value of the expression n.

For example returns:
+1 if the result of n is positive,

0 if the result of n is zero,
–1 if the result of n is negative.

SIN(n)
Calculates the sine of an angle n expressed in radians.

SQR(n)

Returns the square root of the number or expression n.

For example:

`PRINT SQR(6)<ENTER>` returns 2.4494897427831, and
`PRINT SRQ(3*2)<ENTER>` also returns 2.4494897427831.

TAN(n)
This function returns the tangent of the given angle n, which must be expressed in radians.

APPENDIX TWO
BASIC Grammar
Statements

CLS

This statement CLears the Screen, and is equivalent to pressing the CTRL and L keys together.

DEF

Allows particular variables to be allocated to specific types, that is either INTeger, DBL, double precision, SNG, single precision, or STR, string.

For example:

```
DEF STR A,B,C,D
```

defines variables A, B, C and D to be string variables, and thereafter do not have to be defined, that is PRINT A will result in the string value of A being printed. The following short program should indicate the use of this statement:

```
10 DEFSTRA,B,C,D
20 A="FRED":B="AND":C=ALICE":D=" "
30 PRINTA;D;B;D;C
RUN
FRED AND ALICE
```

Indicating that the '$', to define a variable as a string, need not then be used.

Line 10 could also be written as:

```
10 DEFSTRA-D
```

This facility also applies to the numerical variables.

DIM

This statement allows the programmer to allocate specific amounts of memory to be used for a particular variable or variables, and to allocate a given number of subscripts to a particular variable.

That is:

```
DIMA(n)
DIMA(n1),B(n2),C(n3)
```

allocates single variables, called single dimension numeric arrays, for example:

```
DIMA(9)
```

allocates to variable A, 10 subscripts, that is there will be 10 variables called A: A(0), A(1) as far as A(9), indicating that the lowest array element is 0 in a single array. Any attempt to use an element outside the range previously declared will result in a 'Subscript out of range' error. Any attempt to redimension an array later in a program is illegal and will result in a 'Redimensioned array' error.

Arrays can have more than one dimension, but beyond 3, depending on size, the computer will quickly run out of available memory, and produce an 'Out of memory' error.

`DIMD(n1,n2)` is an example of a two dimensional numeric array.

Arrays may also be of a string form, for example:

`DIMna$(1,2)` is an example of a two dimensional string array.

Once an array has been dimensioned, 'information' can then be entered into each element of that array, for example in the two dimensional string array:

```
na$(Ø,Ø)="fred"
na$(Ø,1)="peter"
na$(Ø,2)="john"
na$(1,Ø)="alice"
na$(1,1)="charlotte"
na$(1,2)="hannah"
```

ERASE

Having once dimensioned an array, the occasion may arise where the space allocated is no longer required, or more memory space is needed for it, with the result that the array will either have to be removed or redimensioned.

The statement `ERASE` will allow this to be done by erasing that particular array, and allowing it to be redimensioned if and when required, for example the array 'na$' in the previous example requires more elements, so that more names can be included. To do this use:

```
ERASEna$
DIMna$(3,4)
```

whereupon more elements of the array na$ can now be filled as required.

`ERASE` can be used in exactly the same way as `DIM` in that more than one array can be erased per statement.

END

Indicates the end of the program, beyond which the computer will not go, unless sent there by some specific call, for example `GOTO`, `GOSUB` or `READ`. Closes all files when read by the computer and returns to command mode.

FOR R=n TO m STEP s....NEXT

This statement instructs the computer to execute a statement, or series

of statements, a specified number of times, with a given step between executions. The loop variable R can also be used within the statement or statements, and in consequence will itself be incremented a step each time through the loop. The variable n can have any value, including 0 and negative values, and where the variable m is greater than n, the step is positive, but where it is less than n must be indicated as negative with -s. The executed statements can also include another `FOR....NEXT` loop or loops, the whole then being called a 'nested loop'. Too many nested loops may force an 'Out of memory' error. The statement, or series of statements, is always executed once regardless of the loop variable, and the value of the variable R at the end of the loop is always one step greater than the value of m.

For example:

```
FORR=-2TO6STEP2:PRINT345:NEXT:PRINT"R="R
<ENTER>
```

will produce on the screen:

```
345
345
345
345
345
R= 8
OK
```

and:

```
FORR=1TO5:PRINTR:NEXT<ENTER>
```

will produce:

```
1
2
3
4
5
OK
```

and:

```
FORD=1TO2000:NEXT
```

will produce a reasonable time delay, that is nothing has been programmed to happen within the loop, except for the computer to count from 1 to 2000 round the loop.

Making the loop variable D in the above example an integer will produce an approximately three times faster loop response time.

For example these two loops produce about the same delay times:

```
FORD=1TO1000:NEXT
FORD%=1TO3000:NEXT
```

FRE

This statement when used as PRINTFRE(M) will return the amount of remaining available memory in the computer's RAM for the programmers use, for example:

```
PRINTFRE(M)<ENTER>
```

will return a number indicating the number of free bytes of user memory still available. It is always a good idea to keep a check on this when writing a large program. But compare this to PRINTFRE(M$) in the Appendix dealing with strings, Basic Strings.

For example, try this experiment in command mode:

1. Switch the computer off and on again.

2. You will see that you have 12815 bytes free.

3. Type in : FORR=1TO5ØØ:S=S+6:NEXT<ENTER>

4. Then type in PRINTFRE(M)<ENTER>

5. You should get the number 12787, or thereabouts, which means you have used 12815–12787=28 bytes to RUN number 3.

6. If you now use the screen edit system to change the variable to say D and reENTER the line and then the next line, you will find that the available memory is again reduced. Experiment in this way and you may eventually use up all the memory without even doing one bit of programming!

GOSUB line number and RETURN line number

Sometimes when writing a program a group of lines in that program need to be used in a number of different places within the program. To avoid repeating this same few lines over and over again, they can be separated out into a small sub program called a subroutine, and then called from a number of different places from within the main program.

The format of the subroutine should always be such that its last line is a RETURN statement, though the routine can be RETURNed from any point within it as well. The line that calls the subroutine must say GOSUB line number, but need not necessarily be the first line number of the full subroutine, providing the last line the computer is made to read in the subroutine is a RETURN statement. But the computer does not have to RETURN to the line number following the GOSUB line number statement as it would normally, it can be sent to an alternative line number by the use of the statement 'RETURN line number'.

GOTO line number

Directs the computer to a specific part of the main program, usually used as part of a decision routine.

IF....THEN....ELSE and IF....GOTO....ELSE

These two statements set up a test condition, or a decision making routine,

which are used to control the subsequent action of the computer, for example:

```
IF A=9 THEN PRINT"enough" ELSE PRINT"more
please"
```

The statement following THEN and the statement following ELSE can be any legal statement, or series of statements, for example in this short program:

```
10 FORA=0TO5
20 IFA=5THENPRINT"enough, I have";A;
   "now"ELSEPRINT"I have";A;"so more please"
30 NEXT
```

will print five rows of text stating how many I have and asking for more, and then one row stating that I have 5 and therefore enough.

The value of m cn be changed in the **FOR....NEXT** loop in line 10 and the value of A in line 20 in the **IF....THEN** statement, with subsequent changes to the screen print out.

Changing line 20 to:

```
20 IFA=10THENA=0ELSEPRINT"I have";
   A;"again please"
```

will indicate further the use of the statement, and how the loop variable changes in a **FOR....NEXT** loop, as the loop carries on printing until stopped by the STOP or CTRL/STOP keys, and after the first 0, no more are printed, and neither is the number 10.

The statement can be used without the ELSE statement, whereupon only one action is allowed, that is the one where the statement is true, for example, changing line 20 to:

```
20 IFA=10THENPRINT"finished"
```

will only print out one line.

THEN can be replaced by **GOTO** with or without **ELSE**, but the GOTO must have a line number to go to. For example, change line 20, and add lines 40, 50, 60, 70 and 80 to the previous short program:

```
20 IFA=10GOTO50ELSEGOSUB70
40 END
50 PRINT"finished now"
60 END
70 PRINT"more please"
80 RETURN
```

which should demonstrate the use of the GOTO and GOSUB statements.

Changing lines 20 and 60 to:

```
20 IFA=10THENGOSUB50ELSEGOSUB70
```

```
60 RETURN
```

will produce the same result and show how subroutines can be used in **IF....THEN** statements.

INKEY$

This function will return a one character string from the keyboard. The computer on reading this function in a program scans the 'keyboard buffer' and if a key has been pressed will return that key character as the declared string variable, for example:

```
20 K$=INKEY$:IFK$=""THEN20
```

will stop the computer from proceeding with the program until a key has been pressed, and K$ returned with some character from the keyboard.

The keyboard buffer can, of course, fill up with key presses and store all the characters received until an **INKEY$** function is used, but only one character will be read from the buffer at a time. It is advisable therefore to empty the keyboard buffer before testing it for the key or keys you want by using line 10 as follows:

```
10 K$=INKEY$:IFK$<>""THEN10
20 K$=INKEY$:IFK$=""THEN20
30 IFK$="Y"THENPRINT"You pressed Y"ELSE20
```

This short program will only respond to a capital letter Y being pressed after line 20 has been executed. To allow for both upper and lower case Y's then change line 30 to:

```
30 IFK$="Y"ORK$="y"THENPRINT"You pressed
   the correct key"
```

INPUT

Allows a longer length string to be returned from the keyboard than **INKEY$**.

For example:

```
10 PRINT"Type in your name"
20 INPUT N$:CLS
30 PRINT"Your name is "N$
```

Or can accept numbers:

```
40 PRINT"Type in your age"
50 INPUT N:CLS
60 PRINT"And you are"N
```

INPUT N will not accept letters, whereas **INPUT N$** will accept both.

INPUTs can be grouped together as in:

```
10 PRINT"Type in your name and age"
20 INPUTN$,N:CLS
30 PRINT"Your name is "N$" and you are"N
```

Notice the lack of semi-colons in line 30.

The length of the INPUT string can also be limited to a particular number of characters by using the statement INPUT$(n), as in:

```
10 PRINT"Type in six letters or numbers"
20 N$=INPUT$(6):CLS
30 PRINTN$
```

When RUN the program will clear the screen and then print out the six letters typed in as soon as the last letter of the group of six has been typed, but without line 30, the letters would remain in memory under N$. INPUT, on its own, echoes the letters typed in onto the screen, whereas INPUT$ and INKEY$ do not.

LET

Allows variables to be assigned in memory, but the statement LET is itself not necessary. It is sufficient to say A=10 or A$="cat", and not LETA=10 or LETA$="cat".

LOCATEx,y

This statement allows text to be placed at a particular position on the screen, where, when using the text screen, x=0 to 39, and y=0 to 23, and 0,0 is the top left hand corner of the viewing screen. But note that the computer powers up with only 39 characters per screen row, and must be changed to 40 by the use of WIDTH40<ENTER>, see under WIDTH N at end of this appendix.

The LOCATE statement must be followed by the PRINT statement as in:

```
10 CLS:LOCATE10,10:PRINT"HELLO"
```

The LOCATE statement may also be used with LINE and POINT, refer to the Appendix on graphics.

ON N GOTO and ON N GOSUB

These two statements allow the computer to be sent to a particular line number, ON N GOTO, or to a particular subroutine ON N GOSUB, when a particlar number key is pressed. They are similar to a combination of an INPUT and an IF....GOTO statement, where:

```
10 CLS:INPUTN
20 IFN=1GOTO100
30 IFN=2GOTO200
40 IFN=3GOTO300
50 IFN<>1ANDN<>2ANDN<>3THEN10 is the same as:
10 CLS:INPUTN
20 ON N GOTO100,200,300
30 IFN>3THEN10
```

but is more sophisticated in that it does not require the use of multiple IF....THEN statements.

This short program could be put all on one line of program, as follows:

```
10 CLS:INPUTN:ONNGOTO100,200,300:IFN>3THEN10
```

Remember GOSUB can be used in place of GOTO if subroutines have to be used.

READ, DATA and RESTORE

The use of the **READ** statement allows **DATA** in the form of numbers or strings to be read by the computer and allocated to particular defined variables, for example:

```
10 READN$
100 DATA word
```

or:

```
10 FORR=1TO5:READN$:NEXT
100 DATA WORD,word,WORDS,words,more words
```

The variable name must always coincide with the DATA read, but can be mixed:

```
10 READN$,N
100 DATA word,5
```

or:

```
10 FORR=1TO5:READN$,N:NEXT
100 DATA WORD,5,word,6,WORDS,2,words,88,
    more words,29
```

To include commas in a **DATA** statement requires the DATA to be wrapped in inverted commas:

```
10 READN$
100 DATA "WORDS,WORDS"
```

similarly to use spaces at the end and the beginning of **DATA**

```
10 READN$
100 DATA " 6 words, "
```

It is usual to place the **DATA** at the end of the program, but it can be used more than once by using the **RESTORE** statement, which on its own will **RESTORE** the **READ**ing of the DATA to the first line of **DATA**. To start from any other line of **DATA**, where there is more than one, use the statement '**RESTORE** line number'.

REM

This statement allows instructions to be placed in a program which the computer itself will always ignore. This is useful at the beginning of subroutines, and where the program starts to execute a new sequence to give some indication of what that particular part of the program is doing.

RND(n)

Allows a random number to be generated by the computer depending

on the value of the parameter n used.

`N=RND(1)` will produce a 14 figure decimal number that will be the same each time the program using this function is RUN, but will produce continuously different random numbers within the same program.

`N=RND(Ø)` will produce the same random number each time the program is RUN and the same number throughout the same program.

`N=RND(-TIME)` will produce a different random number each time the number is RUN, and a different number each time it is used in the same program.

To produce random whole numbers the random number must be multiplied by another number and the whole number part taken:

```
N=INT(RND(-TIME)*1Ø)
```

will produce random whole numbers between 0 and 9, as will the use of INT with the other two parameters. And:

```
N=INT(RND(-TIME)*1Ø)+1
```

will produce random whole numbers between 1 and 10.

```
N=RND(INT(-TIME)*15)+2
```

will produce numbers between 2 and 16, the formula to use being:

```
N=RND(INT(-TIME)*X)+Y
```

where Y is the lowest number, and X+(Y–1) is the highest number in the range. To generate a particular range of numbers, first, pick the lowest number you want in your range, Y, and then calculate the value for X from X=(HN–Y)+1. For example, to generate numbers between 25 and 100, Y=25 and X=(100-25)+1=75+1=76, and therefore:

```
N=RND(INT(-TIME)*76)+25.
```

SPC(n)
Used with `PRINT` statements to place blank character spaces before a printed string, for example:

```
N$="I moved 1Ø spaces":PRINTSPC(1Ø)N$
```

STOP

Puts a temporary stop to a program, it does not close files, but does return the computer to command mode. One program can be restarted by `GOTO line number<ENTER>`, using the next line number after the one indicated by the 'Break in line number' statement.

SWAP
This statement allows the values of two different variables to be exchanged. For example:

```
1Ø CLS:Q=1:W=Ø
2Ø PRINT"Q ="Q;"AND W ="W
```

```
30 PRINT"SWAP Q,W":SWAPQ,W
40 PRINT"NOW Q ="Q;"AND W ="W
50 END
```

SWAPs the values of the variables Q and W. This statement works also with strings, and it does not matter which variable you put first, as this program shows:

```
10 CLS:Q$="QWERT":W$="TREWQ"
20 PRINT"Q$ = "Q$;" AND W$ = "W$
30 PRINT"SWAPQ$,W$":SWAPQ$,W$
40 PRINT"NOW Q$ ="Q$;" AND W$ = "W$
50 PRINT"SWAPW$,Q$":SWAPW$,Q$
60 PRINT"AND NOW Q$ ="Q$;" AND W$ = "W$
70 END
```

Please note that the variable separator is a comma, not a semi-colon.

TAB(n)
Used with PRINT statements to place blank character spaces before a printed string, but the calculation is always from the left hand edge of the screen, for example:

```
N$="tabbed 6":PRINTTAB(5)N$;TAB(15)N$"+10"
```

Remember that the first TAB character space is called 0.

VAL(N$)
Allows the computer to translate the numerical value of a string into a numerical variable, but it will not work on strings only containing letters, as it then returns a zero as the value. In any string containing numbers and letters, the letters will be ignored, and only the number translated. For example:

```
N$="67"<ENTER>
PRINTN$<ENTER>
```

produces:

```
67
PRINTVAL(N$)<ENTER>
```

also produces:

```
67
```

but with a blank space in front.

WIDTH N
Allows the screen to have either 40 or 39 characters across a screen row. This is useful if the screen display does not come up to the edge of your TV screen. The computer powers up in WIDTH39, 39 characters per screen row. WIDTH40 will fill to the left hand edge, and will clear the screen at the same time when used, as will a change back to WIDTH39, but does not affect the function key windows displayed at the bottom of the screen.

APPENDIX THREE
BASIC Strings

String variables have a special part of the BASIC language all to themselves, and in this appendix those used on the Spectravideo computer will be explained, with examples where possible.

ASC

ASC stands for American Standard Code II, ASCII, meaning 2 not eleven, and is the standard used to give every character used in the computer a particular code. When we ask the computer to store a particular character it does so, but it does not store the character as such, but its ASCII code number, for example A, which in this case is 65. Therefore:

```
PRINTASC("A")<ENTER>
```

will produce 65, and:

```
PRINTASC("a")<ENTER>
```

will produce 97, as will:

```
PRINTASC("alphabet")<ENTER>
```

as the function only gets the computer to read the first letter of a string. Notice that the letter, word or phrase must always be wrapped in inverted commas, as we are dealing with string variables, not numbers.

```
X$="alpha":PRINTASC(X$)<ENTER>
```

will produce a code 97 as well, as the computer reads the string, X$, and then prints out the first character code of it.

CHR$

This string function is the reverse of ASC, and allows the computer to generate a character from the number given, as follows:

```
PRINTCHR$(65)<ENTER>
```

will produce A, and:

```
PRINTCHR$(97)<ENTER>
```

the lower case 'a'.

This function can be very useful where we would find it difficult to express what we want by using a normal string variable, for example:

```
PRINTCHR$(12)<ENTER>
```

will clear the screen, as CTRL/L clears the screen, and CTRL/L equates to ASCII code 12.

```
PRINTCHR$(7)<ENTER>
```

will produce a short beep sound, as CTRL/G equates to ASCII code 7, and CTRL/G produces a beep from the computer.

You will find that `CHR$(X)` will be used quite a lot in this book, and is especially useful when programming sprites.

FRE

When used as:

```
PRINTFRE(M$)<ENTER>
```

will return the amount of free string memory space remaining, any string variable can be used, and it is called a dummy.

On power up 200 bytes are automatically reserved, but this can be increased by the use of the `CLEAR` statement. To demonstrate, this type in the following program, having first switched the computer off and then on again:

```
10 PRINTFRE(M$)
20 CLEAR300
30 PRINTFRE(M$)
40 FORR=1TO60:A$=A$+CHR$(65):NEXT
50 PRINTFRE(M$)
60 A$=""
70 PRINTFRE(M$)
RUN<ENTER>
200
300
240
300
```

Will show that the computer has 200 bytes of free string memory space available when it is first switched on, line 10. Then more space is reserved by the `CLEAR300` statement on line 20, the number after the CLEAR statement indicating the total amount reserved. String space is then used by incrementing the 'value' of A$ by the letter A, `CHR$(65)`, with the result that string memory space is down to 240 on line 50, having incremented A$ 60 times, 300–60=240. Then A$ is emptied by line 60, A$="", an empty string, and once again string memory is back up to 300 bytes, showing that each character occupied one byte of string memory.

HEX$

This function allows a decimal number to be automatically converted to a hexadecimal number. Hexadecimal numbers use a base of 16, as against decimal which uses a base of 10, and binary which uses a base of 2.

The following short program will show hexadecimal numbers against their decimal equivalents:

```
10 FORR=0TO20:PRINT"HEX "HEX$(R)"=DEC"R:NEXT
```

Changing the values of the loop variables will allow other equivalents to be calculated as required, and to get just one equivalent use:

```
PRINTHEX$(N)<ENTER>
```

where N is the decimal number you wish to convert to hexadecimal, as in:

```
PRINTHEX$(65356)<ENTER>
```

will give the hexadecimal number FF4C.

INSTR

This function, an abbreviation of IN STRing, allows the position of the first occurrence of a particular string in another string to be read, and if required printed out or used elsewhere in a program. For example:

```
10 A$="computer":B$="put"
20 PRINTINSTR(1,A$,B$)
RUN
```

will produce the number 4, indicating that the FIRST occurrence of B$ in A$ starts at character position 4. Changing line 20 to:

```
20 PRINTINSTR(5,A$,B$)
```

will produce the number 0, the position asked for is too late in the string A$ for B$ to be found.

INSTR can be turned into a numerical variable by, for example,

```
C=INSTR(1,A$,B$)
```

The computer does not discriminate between upper and lower case characters when searching for the second string. Line 10 could just as well have been:

```
10 A$="computer":B$="PUT"
```

producing the same results.

LEFT$

This function allows part of one string, from one character to the whole string, to be defined as another string.

For example:

```
10 CLS:A$="computer"
20 C$=LEFT$(A$,7)
30 PRINTC$
RUN
```

will produce the word 'compute', that is the first 7 characters of A$, 'computer', starting at the furthest left hand character.

LEN
Calculates the character length of a string, for example, adding to the program in LEFT$:

```
40 PRINT"C$ contains";LEN(C$);"letters,"
50 PRINT"computer contains";LEN(A$)
```

LEN can also be defined as a variable, for example:

```
60 L1=LEN(A$):L2=LEN(C$)
```

MID$
This function allows a part of one string to be defined as another string.

For example continuing with the same program:

```
70 N$=MID$(A$,4,3)
```

will produce the word put, and:

```
70 N$=MID$(A$,4,3):PRINTN$" has 3"
```

will complete the sentence. MID$(A$,4,3) returns 3 characters of A$ starting at the 4th character.

OCT$
This function is similar to HEX$, but returns a string to a base of eight, the OCTal base. The following line will give some indication of its use:

```
FORR=0TO20:PRINTOCT$(R):NEXT<ENTER>
```

RIGHT$
This function is similar to the LEFT$ function but the count starts from the right hand character of the string.

For example, we can say that:

```
LEFT$(A$,5)+RIGHT$(A$,3)=A$
```

where A$="computer".

Prove it with:

```
80 A$="COMPUTER"
90 PRINTLEFT$(A$,5);RIGHT$(A$,3)
100 PRINTLEFT$(A$,8);RIGHT$(A$,8)
```

SPACE$
Allows a string of a given number of spaces to be defined and used in a program, for example, amend line 100 to:

```
100 PRINTLEFT$(A$,8);SPACE$(5);RIGHT$(A$,8)
```

The maximum number of spaces allowed in one string is 255.

As before, a string variable can hold the SPACE string, as in:

```
SP$=SPACE$(20)
```

Care must be taken to type in SPACE$(20) and not SPACES(20), that is inadvertently typing a capital S in place of a $ symbol.

STR$

This function allows a numeric variable to be converted to a string variable, which can then be used to print a number without its attendant trailing space, for example:

```
PRINT"THE NUMBER";35;"IS 5 X 7"<ENTER.
```

produces:

```
THE NUMBER 35 IS 5 X 7
```

But:

```
PRINT"THE NUMBER";STR$(35);"IS 5 X 7"<ENTER>
```

produces:

```
THE NUMBER 35IS 5 X 7
```

The function is the opposite of VAL, which turns a string representation of a number into the number.

STRING$

This function allows a string to be created that contains up to 255 repeats of one particular character, for example:

```
R$=STRING$(39,187):PRINTR$<ENTER>
```

will produce a line of 39 small heart shapes, useful therefore for producing borders and decorated patterns.

In this example the codes used were the number of characters required followed by the ASC code for the character to be printed.

The function can also use the character itself, for example:

```
R$=STRING$(39,"a"):PRINTR$<ENTER>
```

will print 39 lower case "a"'s. The character required must then of course be placed inside inverted commas. A further variation is to use a string, for example A$, where A$="a", to produce the same result:

```
R$=STRING$(39,A$):PRINTR$<ENTER>
```

Or even where A$="angle" for example, when the computer will use only the first character of the defined string.

VAL

This function instructs the computer to translate the numerical value of a string into a number, for example if:

```
N1$="35":N2$="63"<ENTER>
```

we cannot instruct the computer to add N1$ to N2$ with N1$+N2$ to obtain the answer 98, because the computer is unable to perform numerical computations on strings.

But we can tell the computer to:

```
PRINTVAL(N1$)+VAL(N2$)<ENTER>
```

which will give the numerical answer 98.

We can, of course, ask the computer to add N1$ to N2$ with:

```
PRINTN1$;N2$<ENTER>
```

which produces:

```
3563
```

or even:

```
PRINTN1$;"+";N2$<ENTER>
```

which produces:

```
35+63
```

and also:

```
PRINTN1$;"+";N2$;" =";VAL(N1$)+VAL(N2$)
<ENTER>
```

which will produce the arithmetical equation:

```
35+63 = 98
```

without one numerical variable in sight!

APPENDIX FOUR
BASIC Commands

Every computer has a given number of commands that tell it to carry out some particular operation, the Spectravideo is no exception.

In this Appendix these commands are listed and explained.

AUTO

This command allows line numbers to be automatically generated, with any start line and incremental step required.

For example:

```
AUTO<ENTER>
```

generates the line number starting at line 10 in steps of 10.

The formula AUTO L,S will give any combination required, where L is the start line, and S is the incremental step, for example:

```
AUTO20,5
```

starts at line 20 and continues in steps of 5 lines.

Calling up a previously written line number while in AUTO mode will result in the line number together with an asterisk, «*«, being printed, but with no information about what is on that particular line, however, this does warn the programmer that he is in danger of overwriting the line if he proceeds.

To escape from AUTO mode press the CTRL/STOP keys together.

CONT

This command allows a program to be restarted once it has been STOPped with the STOP command, for example:

```
10 CLS:PRINT"HERE WE GO AGAIN"
20 STOP
30 CLS:PRINT"HALLO AGAIN":FORD=1TO500:NEXT
40 GOTO10
```

will print

```
HERE WE GO AGAIN
Break in 20
```

Typing in:

```
CONT<ENTER>
```

will continue the program, unless any other command has already been

given to the computer such as LIST or RUN.

Any number of characters of the command CONTINUE, which is what CONT stands for, greater than or equal to 4, (CONT, CONTI, CONTIN, etc.), will continue the program.

DELETE

One way to delete a line of program is to type in the line number and then press the ENTER key, for example:

```
35<ENTER>
```

But when you have a lot of lines to remove they can removed in block with the DELETE command.

```
DELETE SL-EL<ENTER>
```

will delete the lines you want, where SL equals the Start Line, or first line of the deletion, and EL is the End Line, or last line. Notice the dash between the parameters, not a comma.

```
DELETE-EL<ENTER>
```

will delete all lines from the beginning of the program up to the EL, or end line indicated.

```
DELETE SL<ENTER>
```

will delete just one line.

Any attempt to delete an undefined line number will result in an 'illegal function call' error.

KEYLIST

This command lists all the commands that have been programmed into the ten function keys, for example:

```
KEYLIST<ENTER>
```

will result in two columns of commands being listed in this order of function key:

1	2
3	4
5	6
7	8
9	10

LIST

This command lists all the BASIC program lines in memory together with the instructions on each one. The following variations are possible:

LIST - produces the full program, but the listing can be paged by pressing the STOP key on and off as required to stop and start the listing.

LIST LS-LE - produces a listing of the required part of the program, LS indicating the first line of the program, and LE the last line.

LIST -LE - will list from the beginning of the program to the line indicated by LE.

LIST LS- - will list to the end of the program starting at the line number indicated by LS.

LIST can also be used inside a program, whereupon once that line is read by the computer, the complete, or as required portion, program is listed, and the computer then returns to command mode.

LLIST

This command lists a program to a printer, and the same rules as in LIST apply. LLIST can also be used from inside a program.

MOTOR

The cassette motor can be turned on and off with this command, both from command mode and from within a program.

MOTORON<ENTER>

switches the cassette motor on, providing one of the cassette recorder keys have been pressed down prior to the command being executed.

MOTOROFF<ENTER>

switches the motor off.

NEW

By typing in this command the complete program will be deleted, together with all the program variables. This is much more drastic than DELETE, which does not remove the variables.

RENUM

Program lines can be renumbered by the use of the RENUM command.

The listing can be renumbered, with alternative steps as required, as follows:

RENUM<ENTER> - will renumber the complete program to start at line 10, in steps of 10.

RENUM SL,EL,IN - will renumber a program with a newline SL, the first line, with EL, the old line, in increments of IN, to the end of the program. Any GOTO line numbers will be automatically renumbered too, thank goodness!

RENUM SL,,IN<ENTER> - will renumber the whole program listing from the beginning with a new start line of SL, in increments of IN.

One way to find out which line numbers have been used before renumbering is to:

AUTO<ENTER>

and then keep your finger on the ENTER key, noting when an asterisk appears against a line number as that line will already have been used.

```
AUTO1,1<ENTER>
```

will of course deal with every possible line as far as line number 65529, any line number beyond this will result in a 'Syntax error' error.

RUN

The most important command you have, it tells the computer to **RUN** your program, and also shows you what mistakes you have made!

You do not have to **RUN** the complete program, you can **RUN** from a particular line number by typing in:

```
RUN LN<ENTER>
```

where **LN** is the line for the program to start from.

SOUND

This command, similar to **MOTOR**, can be used to switch the audio channel of the cassette recorder ON and OFF.

TRON and TROFF

These two commands switch the trace facility of the computer on and off, that is each line, as it is executed, will have its line number displayed in a square bracket at the left hand side of the screen. This is useful when trying to trace a error in a program, as it will indicate all the **GOTO**s you have used, and the results, line number wise, of **IF....THEN** statements!

Type in the following short program for a demonstration of **TRON**.

```
TRON<ENTER>
10 PRINT34
20 PRINT56
30 IF X=1 THEN 10
40 PRINT89
50 PRINT123
60 FORD=1TO5:PRINT99:NEXT
70 PRINT"finished"
RUN
```

This will produce:

```
[10] 34
[20] 56
[30][40] 89
[50] 123
[60] 99
99
99
99
99
[70]finished
```

At the moment there is no action on line 30 due to the **IF....THEN** statement, therefore lines 30 and 40 are reported together. Now type in:

```
X=1<ENTER>
GOTO10<ENTER>
```

will produce:

```
[10] 34
[20] 56
[30][10] 34
[20] 56
[30][10] 34
```

ad infinitum, until the CTRL/STOP keys are pressed, showing the result of the **IF....THEN** statement on line 30.

Typing RUN will of course remove all variables, that is why when we type in the value of X in command mode, we must use **GOTO10**, which **RUN**s the program but does not remove the variables.

Now type in:

```
TROFF<ENTER>
RUN<ENTER>
```

and the tracing facility will be removed.

TRON and **TROFF** can both be used from inside a program. Add the following lines and see what happens:

```
1 TRON
35 TROFF
```

APPENDIX FIVE
BASIC Interrupts

We frequently wish to interrupt the normal flow of a computer program while it is running. To do this we can use the Interrupt commands, some of which appear to allow two things to happen at the same time.

ON ERROR GOTOline number

This command tells the computer which line number of the program to go to when it detects an error in your program.

ON INTERVAL=n GOSUBline number

This command allows the computer to give the impression that two things are happening at the same time, by sending it to a subroutine to carry out some other part of the program, before coming back to where it left off with the main program. It does it all so quickly that the two parts of the program RUN at the same time. The interval 'n' tells the computer how many times per second to carry out the subroutine, 60 is one second in time, therefore:

```
100 ONINTERVAL=30GOSUB1000
```

will send the computer to subroutine 1000 every half a second.

The fastest interval is:

```
100 ONINTERVAL=1GOSUB1000
```

every one 60th of a second.

The facility needs to be enabled at the particular point in the program where it needs to be used, and disabled again when not required, with:

```
10 INTERVAL ON
300 INTERVAL OFF
```

If the facility needs to completely switched off then the command:

```
2000 INTERVAL STOP
```

must be used.

ON KEY GOSUB line number,line number,line number,etc

This command sends the computer to a particular subroutine when a function key is pressed at any time in the program, for example:

```
100 ONKEYGOSUB100,200,300,400
```

will send the computer to subroutine 100 when function key 1 is pressed, subroutine 200 when function key 2 is pressed, etc.

The subroutines do not have to be in numerical order, but the function keys do, for example:

```
100 ONKEYGOSUB200,4000,300,1000,100
```

where function key 1 is subroutine 200, function key 2 subroutine 4000, etc.

To enable the facility use:

```
10 KEYON
```

and to disable:

```
300 KEYOFF
```

and to completely switch off the facility:

```
2000 KEYSTOP
```

The line numbers used are completely fictitious, but indicate the order in which the commands could be used.

ON STOP GOSUBline number

This command must be enabled by `STOP ON`, and disabled with `STOP OFF`, and completely switched off with `STOP STOP`.

It could be used to stop a computer program from ending once it has been RUN, for example, the only way to switch off this short program is to switch off the computer, but it can be stopped by pressing STOP, but then it is only in suspended animation until the STOP key is pressed again:

```
10 ONSTOPGOSUB60
20 STOPON
30 PRINT"Try and stop me"
40 GOTO20
50 END
60 PRINT"TRY AGAIN"
70 RETURN
```

Pressing CTRL/STOP will print:

```
TRY AGAIN
```

in between all the:

```
Try and stop me's
```

ON SPRITE GOSUBline number

This command allows a separate subroutine to be carried out once two sprites have collided with each other, again it must be enabled with `SPRITE ON`, and disabled with `SPRITE OFF`, and completely switched off with `SPRITE STOP`. See the chapter on SPRITES, chapter four, Sprite Characters, for a fuller explanation of this command.

APPENDIX SIX
Screen Modes and Sprites

The Spectravideo, in common with a few other computers, has a number of different screen modes. In MSX BASIC there are three; a text mode, a low resolution mode, and a high resolution mode.

To call up the various screen modes the command `SCREEN` is used as follows:

`SCREEN0` - text mode, 40 x 24, or 39 x 24 characters.

`SCREEN1` - high resolution mode, 256 x 192 pixels.

`SCREEN2` - low resolution mode, 64 x 48 large pixels.

SCREEN0

This mode usually has the function key windows displayed at the bottom of the screen, but these can be removed by the command:

```
SCREEN0,0
```

To replace the function key windows use:

```
SCREEN0,1
```

The statement INPUT will only work in text screen mode, INPUT$ may be a better alternative in other modes.

SCREEN1

This mode is the high resolution mode, and does not have the function key windows, but does have the facility to use sprites. Normally sprites are an 8 by 8 pixel size, and to use these, use either:

```
SCREEN1
SCREEN2
SCREEN1,0
SCREEN2,0
```

The default sprite mode for `SCREEN1` and `SCREEN2` is 0.

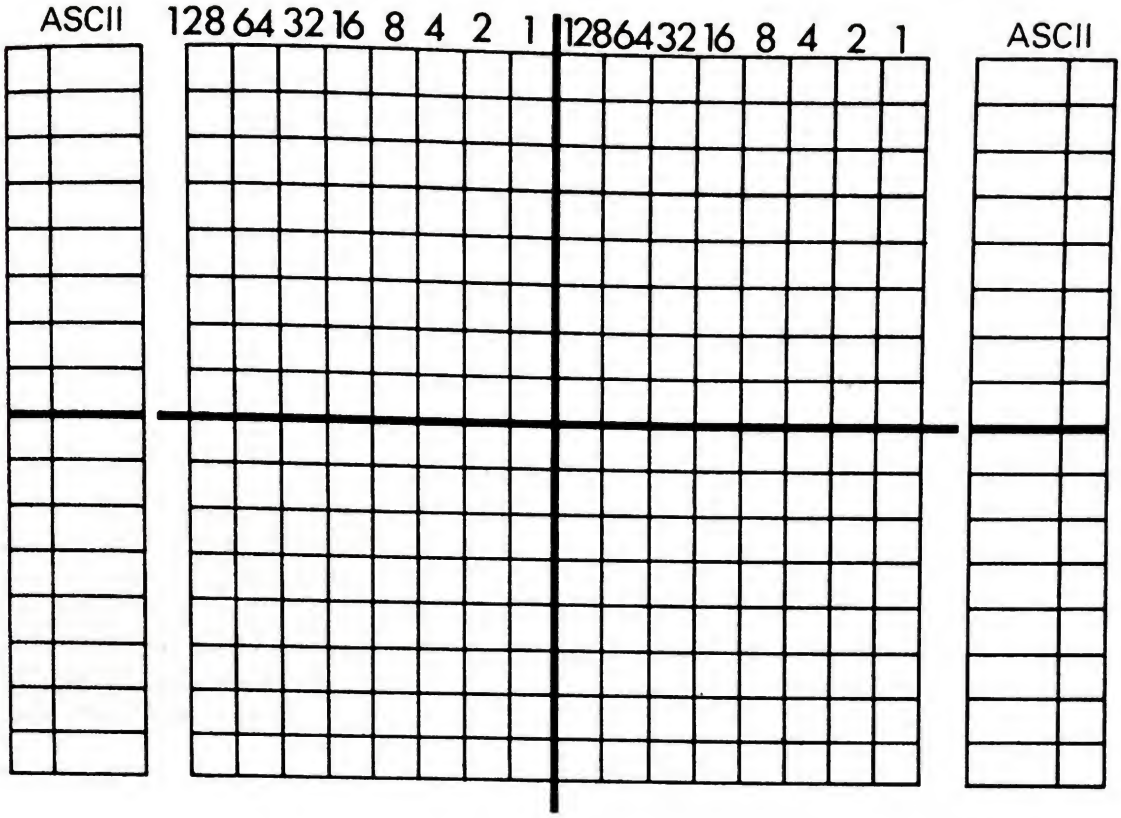
To use the magnified normal sprites use:

```
SCREEN1,1
SCREEN2,1
```

which will produce a 16 by 16 size sprite of the same design as the 8 by 8 without any extra programming.

To use the magnified reprogrammed 16 by 16 sprites use:

```
SCREEN1,2
SCREEN2,2
```



But these sprites can be made smaller by only programming a smaller part of them. A `SCREEN1,2` and `SCREEN2,2` magnified sprite requires 32 bytes of DATA, but can be made smaller by only programming a portion of the 32 bytes starting at byte number 1.

In other words:

bytes 1 to 8, (0 to 7), fill the top left hand pixel area,
bytes 1 to 16 the complete left hand side of the sprite area,
bytes 1 to 24 the complete left hand side plus the top right hand portion.

It is advisable to colour the text or foreground, background and border of a screen before calling the screen, for example:

```
COLOR1,14,14:SCREEN1,Ø
```

not:

```
SCREEN1,Ø:COLOR1,14,14
```

`PUTSPRITE` and `SPRITE$(S)` are two other sprite commands used in MSX BASIC. For a fuller explanation of the sprite facility see chapter four, Sprite Characters.

APPENDIX SEVEN
BASIC Graphics commands

Various chapters within this book explain the use of the graphics commands and statements in great detail and this appendix will list them with a short description of their use.

CIRCLE

This command allows a circle or ellipse to be drawn on either the high resolution or low resolution screens. Parts of a circle or ellipse can also be drawn by defining the extent to which the circumference is drawn. The circle cannot be automatically filled with colour, the `PAINT` command must also be used. The colour of the circumference can be either defined or not, if it is not defined, then the default declared text or foreground colour in the `COLOR` command will be used. Care must be taken to use the same colour in a `PAINT` command as that used for the circumference to avoid the whole screen being filled with colour. The shape of the ellipse can be changed by defining the aspect ratio of the circumference drawn, as can the size of the shape by defining the radius, but this measurement will be used by the computer to measure the horizontal radius only.

The command is:

```
CIRCLE (x,y),z,c,a,b,v/h
```

where:

x = the horizontal location of the centre of the shape,

y = the vertical location of the centre of the shape,

z = the horizontal radius of the shape,

c = the colour of the circumference,

a and b = those fractions of 2PI that determines the amount of the circumference to be drawn,

v/h = determines the aspect ratio by use of the two parameters v/h.

COLOR

Allows the colour of the text, in screen 0, the foreground in screens 1 and 2, the background, and the border to be defined.

The command is:

```
COLORf,ba,bo
```

where:

f = the text or foreground colour,

ba = the background colour,

bo = the border colour.

Notice the spelling of the command 'COLOR', 'COLOUR' will produce a 'Syntax error in line N'.

Any number of parameters may be defined by the command COLOR, but the correct number of commas must be used so that the computer can recognise which colour needs to be changed, for example:

```
COLOR,,15
```

will colour the border white.

```
COLOR,15,15
```

will colour the whole screen white.

```
COLOR,15
```

will colour just the viewing screen white and leave the border in the previous defined colour.

```
COLOR15
```

will colour the text white.

```
COLOR15,,15
```

will colour the text white and the border white, but leave the background in the previous defined colour.

All these commands must be used in a program, and with the exception of the text command COLORn, will have no noticeable effect in command mode.

```
COLOR3<ENTER>
```

will have an immediate effect on the colour of the text, and is a good thing to remember if your program listing suddenly disappears for no apparent reason!

DRAW

This is called the Graphics Macro Language, and enables various shapes to be drawn by the use of DRAW strings. This is fully explained in chapter five, Draw Strings.

GET and PUT

These two commands allow particular areas of the high and low resolution screens to be stored in memory, with GET, and then redisplayed in different areas of the screen, with PUT.

To use GET an area of string memory, an array, must first be reserved with a DIM statement, and then the area to be stored defined by a diagonal across the required area with the command:

```
GET(x1,y1)-(x2,y2),A
```

where x1 and y1 indicate the top left hand corner of the required area, and x2 and y2 the bottom right hand corner of the area. The code A is the name of the array into which the stored pixels have been placed.

To use PUT the top left hand corner of the position where the stored area is to be placed must be defined by:

```
PUT(x,y),A,O
```

where x and y are the two positions, A the array name, and O the particular operation that determines the colour that the PUTted area will be displayed in. This can be either:

PSET - the same colour as the GETted area.

PRESET - the complementary colour of the GETted area.

AND - compares logically the two colours, the array colour and the PUTted area screen colour, to determine the colour used for the displayed area.

OR - same as AND, but different logic.

XOR - same as AND, but again different logic to either AND or OR.

For a fuller explanation of GET and PUT see the chapters on graphics.

PAINT

This command instructs the computer to fill with a particular colour any completely enclosed screen shape. Trying to fill or PAINT non-enclosed shapes will result in the whole screen being filled with colour, as will using a different PAINT colour code to that used to draw the shape in the first place.

The command is:

```
PAINT(x,y),C
```

where x and y are the locations where the PAINTing has to start, which should always be well inside the shape to be filled, and C is the PAINT colour to be used. If the code C is omitted, the default or previously defined foreground colour will be used.

POINT

Allows the colour of a particular screen pixel to be read by the computer. The command is:

```
POINT(x,y)
```

where x and y are the two loactions that define the position on the screen of the pixel to be read.

PSET and PRESET

These two commands allow individual pixels in SCREEN1 to be coloured, or groups of low resolution pixels to be coloured in the same colour in SCREEN2.

Basically PSET will colour a pixel to a new colour with:

```
PSET(x,y),C
```

where x and y are the pixel's location, and C the colour to be used, and

```
PRESET(x,y)
```

will set the pixel back to the background colour.

But the use of the two commands is more complicated than this. In fact the same things can be done with both commands, as PRESET can also have a defined colour with

```
PRESET(x,y),C
```

A fuller explanation of these two commands is given in the chapter entitled Pixel Set.

LINE

This command allows lines to be drawn in either the SCREEN1 or SCREEN2 modes, from one pixel location to another.

The command is:

```
LINE (x1,y1)-(x2,y2),C,BF
```

where x1 and y1 are the start location of the line and x2 and y2 the end location.

C is the colour of the line.

B tells the computer to draw an enclosed rectangle, and F to fill it with the colour defined as C.

To draw an enclosed rectangle the diagonal must be defined by the two locations.

Continuous lines can be drawn by using continuous LINE commands, for example:

```
LINE(x1,y1)-(x2,y2):LINE-(x3,y3):LINE-(x4,y4)
```

APPENDIX EIGHT
BASIC Sound

MSX BASIC and the Spectravideo have two main ways of programming sound.

The first, using the command PLAY, is designed to produce musical sounds, as the name implies.

The second, programmed by the command SOUND, is the command used to produce special sound effects, and although it is quite capable of producing music, the programming for it tends to be more complicated.

Two other sounds can be produced by the computer, both indicatory sounds, which are used to inform the user that something has happened, and both of these can be simulated by the SOUND command, They are the BEEP sound, which indicates to the user that a command has been executed, and CLICK is the sound used to indicate that a key has been pressed.

BEEP

This sound, while being produced by the computer itself automatically as required by the operating system, can also be used in a program, or in command mode to produce the BEEP sound.

The command is:

```
BEEP<ENTER>
```

in command mode, which produces a reasonably high frequency, short duration sound.

In program mode the command is used as normal:

```
10 BEEP
RUN
```

which will again produce the same sound.

To increase the length of the BEEP sound the command can be repeated in two ways, first:

```
10 BEEP:BEEP:BEEP:BEEP:BEEP
RUN
```

will produce a longer duration sound, and:

```
10 FORR=1TO10:BEEP:NEXT
RUN
```

will produce a vibrating sound, the length of which is determined by the loop variable.

Interesting effects can be simulated with the simple BEEP command,

for example:

```
10 BEEP
20 GOTO10
RUN
```

This short program produces a continuous whistle, with a vibrating rumble underneath, but can be used to simulate morse code by repeatedly pressing the STOP key. To stop the program press CTRL/STOP.

CLICK

This sound is produced by the operating system each time a key is correctly depressed. The sound can be switched off with:

```
CLICKOFF<ENTER>
```

and on again with:

```
CLICKON<ENTER>
```

It can also be used in a program to silence the keyboard with:

```
10 CLICKOFF
```

or re-enabled with:

```
100 CLICKON
```

PLAY

This command is the music makers command, and allows music to be simply written in the form of strings, in a similar manner to the **DRAW** command. The Spectravideo has seven octaves of sound, and the ability to play both sharpened and flattened notes. The computer has three channels of sound, which can be played individually or in harmony, and the volume of each can be individually programmed. Although notes of different lengths can be used, the tempo of the overall piece of music can also be changed to suit the mode of the player, and to a degree some control of the envelope of sound produced is available.

A complete description of the **PLAY** command is available in chapter eight, Play Strings.

SOUND

As mentioned before this is the command used to produce special effects as well as musical sounds.

The **SOUND** command has thirteen registers, which allow all three channels to be individually programmed, either on their own or in harmony.

Noise is available, together with a complete envelope facility with eight different basic envelope shapes. Each channel can be individually programmed as regards volume, and a facility exists to make a particular noise frequency predominant over the rest.

An introductory explanation of the **SOUND** facility is available in chapter nine, Synthetic Sounds.

INDEX

Main reference to a BASIC call is enclosed in < >

A NEW COMPUTER AND A POWERFUL OPERATING SYSTEM

---that's the Spectravideo range, with MSX BASIC.

But, because they are so new, you'll need a book to explain how to get the most from them.

The first part of this book deals with editing and debugging of programs -- a subject hardly dealt with in the manual. After that, you'll quickly find how to write programs for education or entertainment, writing text, graphics, sprites and sound.

A novel feature of this book is that each new idea is presented as a problem to be solved. The program which solves the problem is then explained in detail but, you can of course, write your own program first and see how it compares!

Several appendices are included, which cover the three screen modes, sprites and sound.

With this book you'll quickly see how to produce interesting and entertaining programs in the shortest possible time.

Sigma Press
5 Alton Road
Wilmslow
Cheshire
SK9 5DY

Price £7.95

ISBN 0 905104 89 7